UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS - UFR Sciences
École Doctorale STIC

THÈSE

pour obtenir le titre de
Docteur en Sciences
de l'Université de Nice-Sophia Antipolis

Discipline: Informatique

présentée et soutenue par
Ignacio ARAYA

Exploitation des sous-expressions communes et de la monotonie des fonctions pour les algorithmes de filtrage sur intervalles

(Exploiting Common Subexpressions and Monotonicity of Functions for Filtering Algorithms over Intervals)

Thèse dirigée par Bertrand NEVEU et Gilles TROMBETTONI
et préparée à l'INRIA Sophia-Antipolis, projet COPRIN

Soutenue le 3 mars 2010

Rapporteurs:

M. **Benhamou, Frédéric**	Professeur à l'Université de Nantes
M. **Jaulin, Luc**	Professeur à l'ENSIETA, Brest
M. **Michel, Laurent**	Professeur associé à l'Université du Connecticut, USA

Jury:

M.	**Benhamou, Frédéric**	Professeur à l'Université de Nantes
M.	**Jaulin, Luc**	Professeur à l'ENSIETA, Brest
M.	**Neveu, Bertrand**	Ingénieur en chef des Ponts et Chaussées, ENPC, INRIA
M.	**Rueher, Michel**	Professeur à l'Université de Nice Sophia Antipolis
Mme	**Sam-Haroud, Djamila**	Professeur assistant à l'EPFL, Lausanne
M.	**Trombettoni, Gilles**	Maître de conférences à l'Université de Nice Sophia Antipolis

Abstract

This thesis deals with the solving of nonlinear systems of equations/constraints by interval-based methods. We present four contributions, all of them aiming at improving constraint propagation algorithms. Constraint propagation contracts the variable domains of individual constraints with a revise procedure and propagates the changes to the rest of the system.

Our first two contributions are related to monotonicity of functions and to the dependency problem that occurs when a variable appears several times in a function f. In this case, interval-based methods can generally compute only an approximation of the exact range of f. In the same way, these methods cannot contract optimally the variable domains related to an individual constraint. First, we propose a new `Mohc-Revise` algorithm that computes the optimal contraction of variables (w.r.t. an individual constraint) when the function is monotonic. The second contribution is an *Occurrence Grouping* technique that transforms a function f into an equivalent function f_{og}, such that the range of f_{og} can be better approximated by using the monotonicity property.

The third contribution is related to the common subexpression elimination technique (CSE). CSE is a well-known technique mainly used in code optimization. It basically consists in replacing each subexpression $g(X)$ shared by two or more expressions by an auxiliary variable v and adding the new constraint $v = g(X)$. In this thesis we prove that the use of CSE techniques in interval analysis as a preprocessing can improve the filtering power of constraint propagation algorithms.

Finally, we propose a new partial consistency focusing on well-constrained subsystems of size k. Contracting these subsystems can bring additional filtering compared to global contractors (like interval Newton) and revise procedures used in constraint propagation.

Résumé

Cette thèse porte sur les méthodes d'intervalles pour la résolution de systèmes de contraintes non linéaires. Nous présentons quatre contributions, visant toutes à améliorer les algorithmes de propagation de contraintes. Ces algorithmes contractent les domaines des variables d'une contrainte avec une procédure de révision (*revise*) et propagent les modifications dans le reste du système.

Nos deux premières contributions sont liées à la monotonie des fonctions et au problème de dépendance qui se produit quand une variable apparaît plusieurs fois dans une fonction f. Dans ce cas, les méthodes d'intervalles ne peuvent généralement calculer qu'une approximation de l'image de f. De même, ces méthodes ne peuvent pas contracter de manière optimale les domaines des variables impliquées dans une contrainte.

Premièrement, nous proposons un nouvel algorithme `Mohc-Revise` qui calcule la contraction optimale des variables (par rapport à une contrainte individuelle) quand la fonction est monotone. La seconde contribution de regroupement d'occurrences (*Occurrence Grouping*) est une technique qui transforme une fonction f en une fonction équivalente f_{og}, telle que l'image de f_{og} peut être mieux approximée en utilisant la monotonie de f_{og}.

La troisième contribution est liée à la technique d'élimination de sous-expressions communes (CSE). CSE est une technique bien connue principalement utilisée en optimisation de code. Elle consiste essentiellement à remplacer chaque sous-expression $g(X)$, partagée par deux ou plusieurs expressions, par une variable auxiliaire v et à ajouter la nouvelle contrainte $v = g(X)$. Dans cette thèse, nous prouvons que le prétraitement par des techniques de CSE peut améliorer le filtrage des algorithmes de propagation de contraintes.

Enfin, nous proposons une nouvelle consistance partielle basée sur les sous-systèmes bien-contraints de taille k. Contracter ces sous-systèmes peut apporter du filtrage supplémentaire par rapport aux contracteurs globaux (comme Newton intervalles) et par rapport aux procédures de révision utilisées dans la propagation de contraintes.

Acknowledgements

First of all, I would like to express my gratitude to my supervisors and friends, Bertrand Neveu and Gilles Trombettoni, whose expertise and understanding have helped me to a great extent in completing this thesis and becoming a better researcher. I have also appreciated their availability, knowledge and assistance in writing articles, this thesis and so on.

I would like to thank Frédéric Benhamou, Luc Jaulin and Laurent Michel for having accepted to be the reviewers of my thesis. Their comments and remarks have been significant for improving the last version of this thesis. I would also like to thank Michel Rueher and Djamila Sam-Haroud for accepting to be part of my thesis jury.

A very special thanks goes to all the members of the COPRIN project, in particular Jean-Pierre Merlet for having accepted me in his team. His experience and remarks have been useful during my stay. Thanks also go to the other members of the team (David Daney, Odile Pourtallier, Yves Papegay, Julien Hubert, Raphaël Pereira, Mandar Harshe, Guillaume Aubertin, Christine Claux) for having made pleasant the three years I have spent at INRIA.

I would like to thank the Chilean (and Hispanic) people who were always ready to help a compatriot in times of difficulty. Special thanks go to Carlos Grandón, Marcela Rivera, Cristian Ruz and Gabriela Gallegos.

I would also like to thank my parents for the support they provided me through my whole life. Finally, I must acknowledge my wife, Marian, for her support, encouragement and love.

This work was partially supported by the National Commission for Scientific and Technological Research (CONICYT), Chile.

Contents

Chapter 1

Introduction

Systems of nonlinear constraints arise in many domains of practical importance such as engineering, mechanics, medicine, chemistry, and robotics. Solving a system consists in finding all the elements in the search space that satisfy all the constraints of the system at the same time (i.e., the solutions of the system). There are several approaches for solving these problems. We may divide them into three main categories:

1. Symbolic computation methods that solve systems of equations in an exact way.
2. Continuation methods that are effective for problems with algebraic constraints for which the total degree is not too high.
3. Interval-based methods that are generally robust (i.e., reliable) but tend to be slow.

Contrarily to numerical methods, like continuation and interval-based methods, symbolic computation methods are able to obtain the set of solutions of a system in an exact and formal way. The main drawbacks are related with its limited application (only algebraic systems) and the high complexity of existing algorithms (often exponential). The Gröbner basis methods [Buchberger, 1985] are one of the most classical symbolic approaches. A Gröbner basis for a system of polynomials is an equivalent and simpler form of that system, from which information about the roots of the original system can be derived. The Gröbner basis method can be seen as a nonlinear generalization of the Gaussian elimination for linear systems. The resultant elimination method [Gelfand et al., 1994] is based on the determinant theory. It consists basically in solving multipolynomial systems by constructing a univariate polynomial expression such that all the roots of the original multipolynomial system are represented by roots of the constructed univariate polynomial. The continuation methods, also referred to as homotopy methods (see [Hirsch et al., 2007], [Martinez, 1994], [Nielson and Roth, 1999]), basically transform gradually an *initial* system of equations into the system to be solved. The solutions of the initial system are known. At each step i, the current system is solved to find a starting solution for the next system. Each current system is generally solved by a (local) Newton-based method.

Classical interval-based methods for solving systems of constraints are mainly based on the branch & prune technique. The domains of variables (search space) are represented by a set of intervals (*box*). The branching/bisection consists in dividing the box into two subboxes (i.e., the problem is divided into two subproblems). The pruning/contraction consists in eliminating values from the bounds of the current box that do not satisfy the constraints (the entire box is sometimes eliminated). Contraction methods mainly come from two communities: interval analysis (e.g., the interval Newton method) and

constraint programming (e.g., AC3-like constraint propagation algorithms). Bisections and contractions are interleaved in the solving process generating a search tree. When a leaf of the search tree reaches a given precision ϵ, the corresponding box is returned as a solution. Finally, the set of returned boxes contains all the solutions of the system.

In this thesis we propose some sophisticated interval-based methods to solve systems of nonlinear constraints. These new methods are focused on the contraction algorithms issued from the constraint programming community. *Constraint propagation* algorithms call an atomic *Revise* procedure for reducing the domains of variables involved in one constraint c. The procedure eliminates, from the bounds of the intervals, set of values not satisfying c. The domain reductions are then propagated to all the system. Different `Revise` procedures (e.g., `HC4-Revise` described in Section 3.2.3.4, `Box-Revise` described in Section 3.2.3.5) have been proposed. However, none of them is able, in general, to find the smallest box containing all the solutions of c (*hull-consistency*) in polynomial time.

More sophisticated algorithms (e.g., `3B`) split the domain of the variables one by one. Based on a refutation principle, they try to prove the inconsistency of the obtained subdomains by applying a constraint propagation algorithm on the entire system.

We have identified two major problems that hinder the effectiveness of constraint propagation. One is the so-called *dependency problem* related to multiple occurrences of variables in an expression: when a variable appears twice or more in an expression f, the interval methods compute an overestimated image of the variable domains under f.

Example 1 *Consider the following trivial equation involving two variables with domains* $[x] = [-1, 1]$ *and* $[y] = [5, 5]$*:*

$$y = x - x$$

Clearly the image of the expression $x - x$ *is* 0*, implying* $y = 0$*, for any value of* x *over the reals. However, using interval arithmetic the value of* y *is significantly overestimated (*$[x] - [x] \supset 0$*). The new domain of* y*, computed using the classical (natural) evaluation by intervals, is* $[y] = [5, 5] \cap ([-1, 1] - [-1, 1]) = [-2, 2]$.

All the existing interval extensions are impacted by the dependency problem. One of them, the extension by monotonicity (see Section 2.4.2), is able to compute the *optimal image* when f is monotonic w.r.t. all its variables (provided that f is continuous in all the search space). However, existing `Revise` procedures for the constraint $f(X) = 0$, even if f is monotonic, are still non-optimal.

The second difficulty is related to the locality scope of constraint propagation. This *locality problem* also prevents constraint propagation algorithms from computing the smallest box containing all the solutions of a system (global hull-consistency).

Example 2 *Consider for instance the following simple linear system:*

$$\begin{aligned} x + y &= 7 \\ x + y + z &= 12 \end{aligned}$$

With domains of variables $[x] = [0, 5]$, $[y] = [0, 10]$ *and* $[z] = [0, 10]$*. Enforcing the global hull-consistency by hand is trivial. As* $x + y$ *is equal to 7, using the second constraint,* $z = 12 - 7 = 5$*. Finally, from the first equation we can deduce that* $x \in [0, 5]$ *and* $y \in [2, 7]$.

However, constraint propagation algorithms (even if they use an optimal `Revise` *procedure) cannot enforce the global hull-consistency because they treat each constraint in an independent way. An optimal* `Revise`

procedure reduces, using the first equation, the domain of y to $[2, 7]$. Using the second equation the procedure does not know that $x + y$ is equal to 7, then it cannot reduce the domain of z to $[5, 5]$. The system obtained by the `Revise` *procedure is hull-consistent in the domains but it is far from being global hull-consistent.*

Our contributions in this thesis are related, to a greater or lesser extent, to the dependency and locality problems.

1.1 Contributions

Exploiting common subexpressions

We call *common subexpression* (CS) an arithmetic expression appearing several times in the system. The *common subexpression elimination* method (CSE) consists in replacing the CSs of a system (e.g., $x+y$ in Example 2) by auxiliary variables (e.g., v) adding the corresponding new equations. CSE is a significant technique used in code optimization. It is able to reduce the number of operations of segments of code, thus improving the execution time of a program. In interval analysis, several experts have used CSE techniques, manually or automatically, for reducing the size of the system and the number of operations [Kearfott et al., 1996; Merlet, 2000; Schichl and Neumaier, 2005; Van Hentenryck et al., 1997; Vu et al., 2004].

Our contributions to this topic are the following:

1. We prove that the benefits of CSE in interval analysis are mainly due to *additional contraction* (see Example 3) rather than just a reduction in the number of operations.

2. We state which CSs are *useful* for bringing an additional contraction (i.e., sums, products and non-monotonic functions).

3. We propose an ad-hoc algorithm, called `I-CSE`, that replaces *useful* CSs implying an additional contraction.

4. The additional contraction explains why we have obtained gains of one or several orders of magnitude on 10 of the 40 tested systems.

Example 3 *When CSE is applied to Example 2 the CS $x + y$ is replaced by an auxiliary variable (v). Thus, we obtain the new system:*

$$\begin{aligned} v &= 7 \\ v + z &= 12 \\ v &= x + y \end{aligned}$$

The addition of the variable v and the equation $v = x + y$ allows contraction propagation algorithms to reduce the domain of z to $[5, 5]$, thus reaching the global hull-consistency.

Exploiting the monotonicity of functions

Using the well-known evaluation by monotonicity (described in Section 2.4.2), we can compute the optimal image of a box $[B]$ under a function f when f is monotonic w.r.t. all its variables. We can also compute sharper approximations of the optimal image if f is monotonic w.r.t. only some variables.

Our first contributions related to this topic are:

1. We extend the applicability of the evaluation by monotonicity when the function f is *not* monotonic w.r.t. a variable x. In this case, we transform f into a new function f^{og} that replaces some occurrences of x by two auxiliary variables (x_a and x_b with $[x_a] = [x_b] = [x]$) such that the new function f^{og} is monotonic w.r.t. x_a and x_b. The evaluation by monotonicity of f^{og} computes a sharper approximation than the evaluation by monotonicity of f does.
2. For finding *on the fly* a good occurrence grouping, we propose a fast linear program that minimizes a Taylor-based objective function.

The evaluation by monotonicity is commonly used as an existence test for removing boxes that do not satisfying some equation. Less efforts have been devoted to the use of monotonicity in the contraction of variable domains.

We propose a new constraint propagation algorithm `Mohc` which exploits monotonicity of functions. `Mohc` uses monotonic variants of the well-known `HC4-Revise` (used by `HC4`) and `BoxNarrow` (used by `Box`) algorithms for performing better contractions in presence of monotonicities. When f is monotonic w.r.t. every variable with multiple occurrences, `Mohc` is proven to compute the optimal/sharpest box enclosing all the solutions of the constraint (hull-consistency). `Mohc` gives promises to become an alternative to the `Box` and `HC4` algorithms.

Partial consistencies

In the aim of dealing with the locality problem, we investigate the possibility of defining a filtering/contraction algorithm for subsystems of size k.

We propose the revise procedure called `Box-k-Revise` to contract well-constrained subsystems of size k. This algorithm is a solver by itself. Using a branch & prune approach, `Box-k-Revise` is able to find all the solutions of a subsystem with a given precision. It then returns the smallest box enclosing all these solutions of the subsystem. The changes to the variable domains are then propagated to other subsystems thanks to a classical propagation loop. Also, the filtering performed inside a subsystem allows the solving process to learn interesting multi-dimensional branching points, i.e., to split several variable domains simultaneously.

1.2 Organization of the document

Part I presents a state of the art. Chapter 2 is an introduction to interval arithmetic. After introducing the main concepts and notations, we detail the main interval extensions used for computing approximations of images.

Chapter 3 introduces the main interval-based methods used for solving systems of constraints. We briefly describe the algorithms from interval analysis and rigorous global optimization. Then we focus on the

contraction algorithms issued from constraint programming. Finally, we briefly present some background related to our work.

Part II corresponds to our contributions. Chapters 4 and 5 presents an exploitation of the monotonicity of functions for improving contraction. The former presents our new constraint propagation algorithm `Mohc`; the latter describes *occurrence grouping*, a new interval extension that improves the evaluation by monotonicity.

In Chapter 6 we show the benefits of using CSE as a preprocessing technique for improving the performance of propagation algorithms. Then we describe the `I-CSE` algorithm which is dedicated to interval based methods.

Chapter 7 describes the `Box-k-consistency`, a new local consistency related to subsystems of equations. We present an algorithm enforcing this new consistency and the obtained results.

Part I

State of the Art

Chapter 2

Interval Arithmetic

Contents

Basics of interval arithmetic have been used throughout the history of mathematics. Archimedes, for example, in the 3^{rd} century BC calculated lower and upper bounds for π: $223/71 < \pi < 22/7$ using an interval approximation of $\sqrt{3}$ [Archimedes, Before 212 BC].

In the 20^{th} century, Rosalind Cecily Young proposed rules for calculating with intervals and with other subsets of the real numbers in her work *algebra of many-valued quantities* [Young, 1931]. In the 50s, the essential ideas in interval arithmetic were set forth in an independent and almost simultaneous way by Mieczyslaw Warmus in Poland [Warmus, 1956], Teruo Sunaga in Japan [Sunaga, 1958] and Ramon E. Moore in the United States.

Moore wrote in 1966 the book *Interval Analysis* [Moore, 1966] marking the birth of modern interval arithmetic. The book includes both previous works and personal researches performed after 1950. Interval arithmetic is formally defined with the objective of being conservative in the computations of basic operators ($+$, $-$, $\times$, $/$), bounding rounding errors in mathematical computation and thus developing numerical methods that offer reliable results. The basic interval operators applied to intervals (containing imprecision related to the used methods or the floating point computations) generate new intervals preserving the imprecisions and adding new imprecisions related to the operator. We say that the interval arithmetic is *conservative*. By introducing interval arithmetic into existing solving methods we can solve problems without losing solutions.

In this chapter we present the fundamentals of interval arithmetic and the descriptions and definitions of basic interval arithmetic operators and interval extensions of real functions. Interval extensions are commonly used for computing approximations of the range of functions. Finding good approximations is one of the most important issues when interval methods are used for solving system of constraints.

2.1 Basic notions

2.1.1 Intervals

For any pair of real numbers a and b, where $a \leq b$, an **interval** $[a, b]$ is the set:

$$[a, b] := \{x \in \mathbb{R},\ a \leq x \leq b\}$$

$\mathbb{IR}$ corresponds to the set of all the intervals. The elements in $\mathbb{IR}$ are enclosed in brackets ($[x]$). In an interval $[a, b]$, the real number a corresponds to the **left bound** of the interval and b to the **right bound** of it. In the interval $[x]$, the left bound and the right bound are noted resp. $\underline{x}$ and $\overline{x}$. $\texttt{Mid}([x])$ denotes the midpoint of $[x]$, i.e., $\texttt{Mid}([x]) = \frac{1}{2}(\underline{x} + \overline{x})$ and $\texttt{Diam}([x]) = \overline{x} - \underline{x}$ corresponds to the **diameter** of the interval. A **degenerate interval** is an interval with diameter 0 (e.g., $[x] = [a, a]$).

The minimum and maximum absolute real values in an interval are called resp. **mignitude** (denoted $\langle[x]\rangle$) and **magnitude** (denoted $|[x]|$):

$$\langle[x]\rangle := \min_{x \in [x]} |x| = \begin{cases} \min(|\underline{x}|, |\overline{x}|) & \text{if } \ 0 \notin [x], \\ 0 & \text{otherwise.} \end{cases}$$

$$|[x]| := \max_{x \in [x]} |x| = \max(|\underline{x}|, |\overline{x}|)$$

The inclusion relation between two intervals is defined by:

$$[x] \subseteq [y] \Leftrightarrow (\underline{x} \geq \underline{y}) \wedge (\overline{x} \leq \overline{y})$$

A Cartesian product of intervals is named a **box**, and is denoted by $[B]$, $[x_1] \times ... \times [x_k]$ or by a vector $([x_1], ..., [x_k])$. Graphically it can be represented by the set of points inside of a k-dimensional rectangle $(\{\underline{x_1}, \overline{x_1}\}, ..., \{\underline{x_k}, \overline{x_k}\})$ leading to 2^k vertices. For example, the rectangle in Figure 2.1-left represents the box $[B] = ([x], [y])$, where $[x] = [1, 7]$ and $[y] = [3, 6]$. The four vertices of the box are $(1, 3)$, $(1, 6)$, $(7, 3)$ and $(7, 6)$.

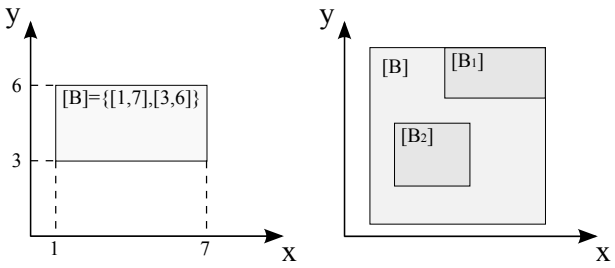

Figure 2.1: On the left side, a graphical representation of a 2-dimensional box $[B] = \{[1, 7], [3, 6]\}$. On the right side, a graphical representation of the inclusion relations $[B_1] \subseteq [B]$ and $[B_2] \subseteq [B]$.

The element in the middle of a box $[B] = [x_1] \times ... \times [x_k]$ is defined by:

$$\texttt{Mid}([B]) = (\texttt{Mid}([x_1]), ..., \texttt{Mid}([x_k]))$$

A **degenerate box** is a box containing only degenerate intervals. Consider two k-dimensional boxes $[B_1] = ([x_1], ..., [x_k])$ and $[B_2] = ([y_1], ..., [y_k])$. The inclusion relations between boxes are given by (see a graphical example in Figure 2.1-right):

$$[B_1] \subseteq [B_2] \Leftrightarrow \forall i = 1, ..., k\text{: } (\underline{y_i} \leq \underline{x_i}) \wedge (\overline{x_i} \leq \overline{y_i})$$

The relations of superiority and inferiority between intervals are defined as follows:

$$[x] < [y] \Leftrightarrow \overline{x} < \underline{y}$$
$$[x] > [y] \Leftrightarrow \underline{x} > \overline{y}$$

A real α can be mapped onto a zero-diameter interval $[\alpha, \alpha]$. Then, we can define relations of superiority and inferiority between an interval and a real number:

$$[x] < \alpha \Leftrightarrow \overline{x} < \alpha$$
$$[x] > \alpha \Leftrightarrow \underline{x} > \alpha$$

Example 4 *Consider the intervals* $[x] = [1, 4]$, $[y] = [0, 2]$ *and* $[z] = [-5, 4]$. *The bounds of* $[x]$ *are* $\underline{x} = 1$ *and* $\overline{x} = 4$*; the diameter of* $[z]$ *is* $\texttt{Diam}([z]) = 4 - (-5) = 9$*; the mignitude and magnitude of* $[x]$ *are* $\langle [x] \rangle = \min(|1|, |4|) = 1$ *and* $|[x]| = \max(|1|, |4|) = 4$*, resp.; the mignitude and magnitude of* $[z]$ *are* $\langle [z] \rangle = 0$ *and* $|[z]| = \max(|-5|, |4|) = 5$*, resp.; the midpoints of* $[x]$ *and* $[y]$ *are resp.* $\texttt{Mid}([x]) = \frac{1+4}{2} = 2.5$ *and* $\texttt{Mid}([y]) = \frac{0+2}{2} = 1$.

The interval $[x]$ *is included in* $[z]$ *(i.e.,* $[x] \subseteq [z]$*). The interval* $[x]$ *is less than* $[5, 7]$ *and greater than* 0 *(i.e.,* $[x] \leq [5, 7]$ *and* $[x] > 0$*). The box* $([1, 3], [0, 1], [-1, 2])$ *is included in the box* $([x], [y], [z])$*, i.e.,* $([1, 3], [0, 1], [-1, 2]) \subseteq ([x], [y], [z])$.

2.1.2 The hull operator

The **union** ($\cup$) of two or more intervals corresponds to the set of the real numbers included in at least one interval, i.e.:

$$[x_1] \cup ... \cup [x_n]; = \{x, x \in [x_1] \vee ... \vee x \in [x_n]\}$$

The union of boxes corresponds to the set of vectors included in at least one box, i.e.: $[B_1] \cup ... \cup [B_n] = \{(x_1, ..., x_k), (x_1, ..., x_k) \in [B_1] \text{ or } ... \text{ or } (x_1, ..., x_k) \in [B_n]\}$

Example 5 *Consider the intervals* $[x] = [2, 6]$, $[y] = [-1, 3]$ *and* $[z] = [5, 8]$. *Then:*

$$[x] \cup [y] = [-1, 6] \quad (2.1)$$
$$[x] \cup [z] = [2, 8] \quad (2.2)$$
$$[y] \cup [z] = [-1, 3] \cup [5, 8] \quad (2.3)$$

Observe that the result in 2.3 is not an interval. The *set union* operation is not closed in $\mathbb{IR}$.

In intervals, the *hull operation* is used generally instead of the union operation. Consider a set of intervals $S_{[x]}$. The **hull** of this set ($\texttt{Hull}(S_{[x]})$) corresponds to the *smallest interval* including every interval in $S_{[x]}$ (i.e., $[x] \in S_{[x]} \Rightarrow [x] \subseteq \texttt{Hull}(S_{[x]})$). Thus, the hull of the interval set $S_{[x]}$ can be computed as follows:

$$\texttt{Hull}(S_{[x]}) = \left[\min_{[x] \in S_{[x]}} (\underline{x}), \max_{[x] \in S_{[x]}} (\overline{x}) \right]$$

Replacing the union operation in Expression 2.3 by the `Hull` operation we obtain $\texttt{Hull}([y], [z]) = [-1, 8]$.

The hull of a set of boxes $S_{[B]}$ ($\texttt{Hull}(S_{[B]})$) corresponds to the smallest box including every box in $S_{[B]}$ (i.e., $[B] \in S_{[B]} \Rightarrow [B] \subseteq \texttt{Hull}(S_{[B]})$). Consider $[x_{ij}]$ the i^{th} interval of a box $[B_j]$; the hull of a set of n k-dimensional boxes $S_{[B]} = \{[B_1], ..., [B_n]\}$ provides the following box:

$$\texttt{Hull}(S_{[B]}) := \texttt{Hull}([x_{11}], ..., [x_{1n}]) \times ... \times \texttt{Hull}([x_{k1}], ..., [x_{kn}])$$

Figure 2.2 represents graphically the hull operator.

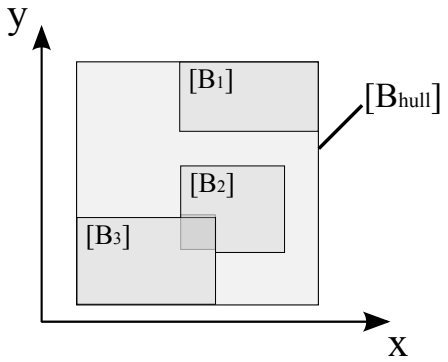

Figure 2.2: Hull of boxes. $[B_{hull}]$ represents $\texttt{Hull}([B_1], [B_2], [B_3])$

Example 6 *Consider the boxes* $[B_1] = ([0, 3], [1, 7])$, $[B_2] = (-1, 8)$ *and* $[B_3] = ([1, 5], [2, 2])$. *The hull of the three boxes is computed:*

$$\begin{aligned} \texttt{Hull}([B_1], [B_2], [B_3]) &= (\texttt{Hull}([0, 3], -1, [1, 5]), \texttt{Hull}([1, 7], 8, [2, 2])) \\ \texttt{Hull}([B_1], [B_2], [B_3]) &= ([-1, 5], [1, 8]) \end{aligned}$$

Consider $[B_{hull}] = \texttt{Hull}([B_1], [B_2], [B_3])$. *Just as we expected* $[B_1]$, $[B_2]$ *and* $[B_3]$ *are included in* $[B_{hull}]$ *(i.e.,* $[B_1] \subseteq [B_{hull}]$, $[B_2] \subseteq [B_{hull}]$ *and* $[B_3] \subseteq [B_{hull}]$*).*

2.1.3 Image of intervals

One of the most important issues in interval methods is to find good approximations of the image of real functions. As intervals and boxes are mainly used for representing domains of variables, we define the image of a box $[B]$ under a function f by:

$$\mathcal{I}f([B]) = \{y \in \mathbb{R}, \exists (x_1, x_2, ..., x_n) \in [B], y = f(x_1, x_2, ..., x_n)\} \tag{2.4}$$

The image of $[B]$ under f is the set of all possible outputs obtained when the function is evaluated at each element of the box $[B]$. For example, the image of the interval $[x] = [-1, 2]$ under the function $f(x) = x^3 - x$ is $\mathcal{I}f([-1, 2]) = [-0.3849, 6]$ (the minimum of f in the interval $[-1, 2]$ is -0.3849, when $x = 1/\sqrt{3}$. The maximum of f is 6, when $x = 2$).

The image of an interval under a continuous function is an interval. In the general case the image can be represented as a set of intervals. We call **optimal image** the sharpest interval containing $\mathcal{I}f([B])$ (i.e., the hull of the image). $\texttt{Hull}(\mathcal{I}f([B]))$ denotes the optimal image of $[B]$ under the function f.

Example 7 *Consider the non-continuous function* $f(x, y) = x/y$ *with domain* $[B] = [-2, -1] \times [-1, 1]$. *The image of* $[B]$ *under* f *is* $\mathcal{I}f([B]) = (-\infty, -1] \cup [1, +\infty)$. *The optimal image is* $\texttt{Hull}(\mathcal{I}f([B])) = (-\infty, +\infty)$.

2.2 Interval Arithmetic

In this thesis we work with mathematical expressions/functions/constraints that are composed of a set of basic operations. This set includes binary arithmetic operations ($+$, $-$, $\times$, $/$) and unary functions such as *sin*, *cos*, *exp* and *log*. For example:

$$x/y + \exp(sqr(z \times x)) - \sin(x)/2$$

In this section we use intervals and *interval operators* for building interval expressions and interval functions. An **interval function** $[f]$ is analogous to a function over the reals, i.e., it is a function that receives a set of intervals as input and returns an interval as output ($[f] : \mathbb{IR}^k \to \mathbb{IR}$).

In the following, we explain how to map classical arithmetic expressions to interval arithmetic expressions and functions.

2.2.1 Binary operators

The binary operators of interval arithmetic are trivially extended from the classical arithmetic operators. Consider the basic binary arithmetic operators of addition, subtraction, multiplication and division ($+, -, \times, /$). The corresponding interval operators compute the smallest interval containing the image of the related expression, i.e., if we consider the binary expression $f_\circ(x_1, x_2) = x_1 \circ x_2$ over the reals, then we obtain:

$$[x_1] \circ [x_2] := \texttt{Hull}(\mathcal{I}f_\circ([x_1], [x_2])) \tag{2.5}$$

where $\circ \in \{+, -, \times, /\}$. We say that $[x_1]$ and $[x_2]$ are the **input intervals** of the operator. Using Definition (2.4) we could compute the optimal image of the input intervals under the operator $\circ$:

$$[x_1] \circ [x_2] = \left[\min_{x_1 \in [x_1], x_2 \in [x_2]} x_1 \circ x_2, \max_{x_1 \in [x_1], x_2 \in [x_2]} x_1 \circ x_2 \right] \tag{2.6}$$

All the basic interval operators are defined using (2.6). For instance, as the sum is continuous and monotonic w.r.t. each of its terms, the interval sum operator is given by:

$$[x_1] + [x_2] := [\underline{x_1} + \underline{x_2}, \overline{x_1} + \overline{x_2}]$$

Taking into account properties of continuity and monotonicity, the other operators can be defined in function of the bounds of the input intervals. Consider $[x_1] = [a, b]$ and $[x_2] = [c, d]$.

$$[x_1] - [x_2] := [a - d, b - c] \tag{2.7}$$
$$[x_1] \times [x_2] := [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \tag{2.8}$$
$$1/[x_2] := [1/d, 1/c] \quad 0 \notin [x_2] \tag{2.9}$$
$$[x_1]/[x_2] := [x_1] \times (1/[x_2]) \quad 0 \notin [x_2] \tag{2.10}$$

Observe that, for the moment, the expression related to each operator requires be continuous in all the values of the input intervals (e.g., the division operator is not defined if $0 \in [x_2]$). The *extended interval arithmetic* (described in Section 2.2.7) removes this limitation, offering some advantages for solving a system of constraints.

Example 8 *Consider the intervals* $[x] = [1, 5]$, $[y] = [-2, 3]$ *and* $[z] = [0, 2]$. *Then,*

$$[x] + [y] = [-1, 8] \qquad [y] - [z] = [-4, 3]$$
$$[x] \times [z] = [0, 10] \qquad [y]/[x] = [-2, 3]$$

The interval binary operators and the classical algebraic operators have some equivalent properties (e.g., commutativity of the addition and the multiplication). However, other properties of classical operators are lost (e.g., $[x] - [x] \neq 0$, the distributivity is replaced by the subdistributivity). In Section 2.3.3 we explain the reason of this loss and show some new relaxed properties of interval operators.

Interval operators are well used for computing an approximation of domain images under some function. In this sense, it can be relevant to know how the interval diameter computed by an operator is related with the diameter of the input intervals:

$$\begin{aligned}
\texttt{Diam}([a] + [b]) &= \texttt{Diam}([a]) + \texttt{Diam}([b]) \\
\texttt{Diam}([a] - [b]) &= \texttt{Diam}([a]) + \texttt{Diam}([b]) \\
\texttt{Diam}(\alpha \times [b]) &= |\alpha| \times \texttt{Diam}[b] \quad \alpha \in \mathbb{R} \\
\texttt{Diam}([b]/\alpha) &= \texttt{Diam}[b]/|\alpha| \quad \alpha \in \mathbb{R}, \alpha \neq 0
\end{aligned}$$

Observe that for example, contrarily to what we might think, the diameter of the subtraction of two intervals is not equal to the subtraction of the input interval diameters. So we can deduce, for instance, that $[x]-[x]$ cannot be equal to 0 (except when $\texttt{Diam}(x) = 0$). In Section 2.3.3 we describe the *dependency problem* that is directly related with this observation.

2.2.2 Unary operators

Following the same principle as binary operators, the unary operators of interval arithmetic are defined from classical arithmetic. Consider $f(x)$ a unary arithmetic operator and $[f]([x])$ the corresponding operator over intervals. The operator $[f]$ is defined by:

$$[f]([x]) := \texttt{Hull}(\mathfrak{I}f([x]) = [\min_{x \in [x]} f(x), \max_{x \in [x]} f(x)] \tag{2.11}$$

Like binary operators, the continuity and monotonicity of f can be used for defining the corresponding interval operator $[f]$ in function of the bounds of the input intervals. Consider for example the *square* operator:

$$[sqr]([x]) = [x]^2 := \begin{cases} [\underline{x}^2, \overline{x}^2] & \text{if } [x] > 0, \\ [\overline{x}^2, \underline{x}^2] & \text{if } [x] < 0, \\ [0, max(\underline{x}^2, \overline{x}^2)] = [0, |[x]|^2] & \text{if } 0 \in [x]. \end{cases}$$

Example 9 *Consider the intervals* $[x] = [1, 5]$, $[y] = [0, \pi/4]$ *and* $[z] = [2, 7]$. *Then,*

$$[sin]([y]) = [0, \sqrt{2}/2] \qquad [cos]([y]) = [\sqrt{2}/2, 1]$$
$$[x]^3 = [1, 125] \qquad [log]([z]) = [log(2), log(7)] = [0.693..., 1.946...]$$

2.2.3 Evaluation of interval expressions

By composition of binary and unary operators we can create more complex interval functions. For example:

$$[f]([x],[y],[z]) = [x] + ([y] \times [z])^2 + 4$$

For computing the image of this function it is necessary to represent the expression as a syntactic binary tree (see Figure 2.3). The variables and constants are represented by the tree leaves. The operators are represented by the nodes and the function or expression corresponds to the root node. The computation of the image or **evaluation** of $[f]$ is performed by replacing the variables with the corresponding intervals and applying the operator definitions recursively from the leaves to the root.

Example 10 *Consider the values* $[x] = [1,2]$, $[y] = [2,4]$ *and* $[z] = [-1,1]$. *The evaluation of* $[f]$, *represented on Figure 2.3, performs the following computations:*

$$\begin{array}{lrcl} 1. & [2,4] \times [-1,1] & = & [-4,4], \\ 2. & [-4,4]^2 & = & [0,16], \\ 3. & [1,2] + [0,16] & = & [1,18], \\ 4. & [1,18] + 4 & = & [5,22]. \end{array}$$

Finally the evaluation of $[f]([1,2],[2,4],[-1,1])$ *is* $[5,22]$.

An interval function $[f]$ obtained by mapping a real function f over the intervals is called **natural extension** of f. The evaluation of $[f]$ in a box $[B]$ computes an overestimation of the image of $[B]$ under f. The natural extension is described in Section 2.4.1.

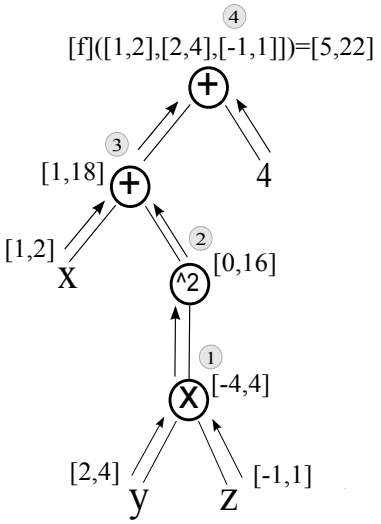

Figure 2.3: The evaluation of $[f]([1,2],[2,4],[-1,1])$ performed in the corresponding binary tree.

2.2.4 Interval gradient computation

Several interval-based methods require a procedure for computing the *interval gradient* (or *interval partial derivatives*) of functions. An **interval partial derivative** of a function f w.r.t. a variable x, denoted

$\left[\frac{\partial f}{\partial x}\right]([B])$ corresponds to an interval enclosing all the partial derivatives of f w.r.t. x in the given box $[B]$, i.e.:

$$\forall X \in [B]: \quad \frac{\partial f(X)}{\partial x} \in \left[\frac{\partial f}{\partial x}\right]([B])$$

2.2.4.1 Symbolic differentiation

A simple method for computing the interval gradient related to a function f consists in symbolically differentiating the function and evaluating it in the box (using the interval arithmetic operators).

Example 11 *Consider the function $f(x,y) = x^2 + 3xy - y$ with intervals $[x] = [-1,1]$ and $[y] = [1,2]$. The symbolic differentiation produces two new expressions: $g_x(x,y) = 2x + 3y$ and $g_y(x,y) = 3x - 1$. Evaluating the two derivatives in the box we obtain the intervals: $[g_x]([B]) = [1,8]$ and $[g_y]([B]) = [-4,2]$. As $[g_x]([B]) > 0$, f is monotonic increasing w.r.t. x.*

2.2.4.2 Automatic differentiation

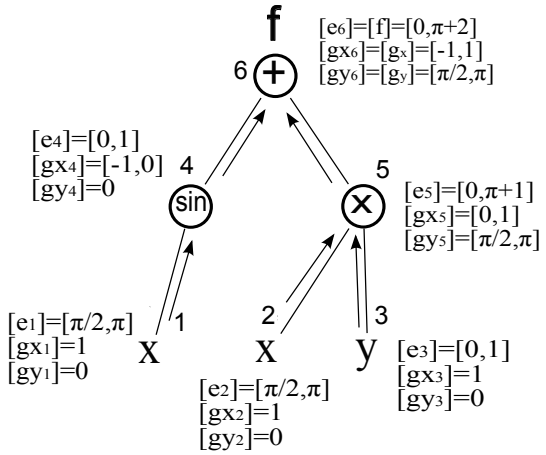

Figure 2.4: Example of automatic differentiation using the forward mode. The interval gradient of $f(x,y) = sin(x) + xy$ is computed.

Another used procedure (which is very often exploited in the new methods presented in this thesis) is **automatic differentiation** (AD). This procedure computes an interval gradient without obtaining the corresponding analytic expressions. Moore [Moore, 1966] was the first to apply it to interval analysis.

Fundamental in AD is the decomposition of differentials provided by the chain rule. For the composition $f(x) = g(h(x))$ the chain rule yields:

$$\frac{\partial f}{\partial x} = \frac{\partial g}{\partial h} \times \frac{\partial h}{\partial x}$$

There are two modes of performing AD. The *forward mode* traverses the chain rule from right to left, that is, one first computes $\frac{\partial h}{\partial x}$ and then $\frac{\partial g}{\partial h}$. The *reverse mode* traverses the chain rule from left to right. Both use a binary tree representing the expression.

The forward mode

In the *forward mode* the tree is traversed from leaves to root (see Figure 2.4). Each node w_i evaluates the related subexpression using the arithmetic operators ($[e_i]$), and computes the interval partial derivative of the subexpression w.r.t. each variable ($[g_{x_i}]$ and $[g_{y_i}]$). The interval derivatives are computed using *gradient operators* related to each node. The gradient operators are based on differentiation rules (e.g., $\frac{\partial(sin(w_1))}{\partial x} = cos(w_1)\frac{\partial(w_1)}{\partial x}$, $\frac{\partial(w_1+w_2)}{\partial x} = \frac{\partial(w_1)}{\partial x} + \frac{\partial(w_2)}{\partial x}$, etc.) and use the information of the children (evaluation and partial derivatives) for computing the partial derivative of the node. For example, the node w_5 of the figure corresponds to a multiplication of two terms: $w_2 \times w_3$ the partial derivative of w_5 w.r.t. x is computed by a gradient operator: $[gx_5] = [gx_2] \times [e_3] + [e_2] \times [gx_3]$, where $[e_2]$ (resp. $[gx_2]$) is the previously computed evaluation (resp. interval partial derivative) in node w_2; $[e_3]$ (resp. $[gx_3]$) is the previously computed evaluation (resp. interval partial derivative) in node w_3. The gradient of f is obtained at the root node.

The backward mode

In the *backward mode* [Rall, 1981; Speelpenning, 1980], the tree is traversed twice. The first traversal is achieved from the leaves to the root (see Figure 2.5-left). Each node w_i evaluates the related subexpression using the arithmetic operators ($[e_i]$). The second traversal (*reverse pass*) is from the root to the leaves (see Figure 2.5-right). Each node w_i computes, using *backward gradient operators*, the interval partial derivative of f w.r.t. w_i ($[g_i]$). The backward gradient operators are based on differentiation rules (mainly the chain rule $\frac{\partial(f(w_j(w_i)))}{\partial w_i} = \frac{\partial f}{\partial w_j} \times \frac{\partial w_j}{\partial w_i}$, where w_j is the parent of w_i). To compute $[g_i]$ in a node w_i, the backward operator uses the information of the parent and brother nodes. For example, in the node w_2 the backward gradient operator computes the interval partial derivative of f w.r.t. w_2: $[g_2] = [g_5] \times [e_3]$; observe that the operator uses the chain rule in the composition $f(w_5(w_2))$, i.e., $\frac{\partial(f(w_5(w_2)))}{\partial w_2} = \frac{\partial f}{\partial w_5} \times \frac{\partial w_5}{\partial w_2}$ where $\frac{\partial f}{\partial w_5}$ is $[g_5]$ and $\frac{\partial w_5}{\partial w_2} = w_3$ is $[e_3]$. At the end of the second traversal the interval partial derivatives w.r.t. *each occurrence* have been computed. Finally, the interval partial derivatives w.r.t. each variable is obtained by the sum of the partial derivatives of its occurrences (e.g., $[g_x] = [g_1] + [g_2] = [0,1] + [-1,0] = [-1,1]$).

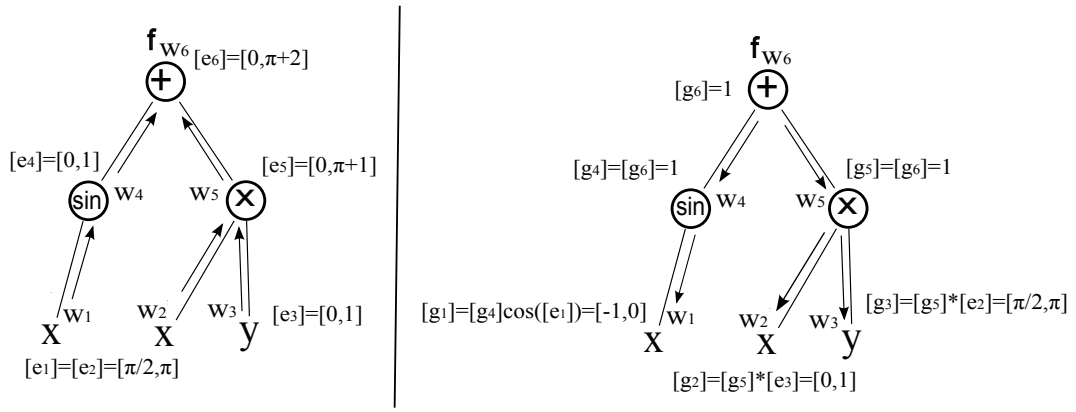

Figure 2.5: Example of automatic differentiation using the backward mode. The interval gradient of $f(x,y) = sin(x_1) + x_1x_2$ are computed.

2.2.5 Interval Hessian matrix computation

The Hessian matrix is the square matrix of second-order partial derivatives of a function. Consider the function $f(x_1, ..., x_n)$, the Hessian matrix of f is defined by:

$$\mathbf{H_f} = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix}$$

The interval Hessian matrix $[\mathbf{H_f}]$ related to a function f and a box $[B] = ([x_1], ..., [x_n])$ is an interval matrix containing all the values of the Hessian matrix evaluated in each point $(x_1, ..., x_n) \in [B]$, i.e.,

$$\forall (x_1, ..., x_n) \in [B] : \quad \mathbf{H_f}(x_1, ..., x_n) \subseteq [\mathbf{H_f}]([x_1], ..., [x_n])$$

For computing the elements of the interval Hessian matrix (i.e., intervals enclosing each second-order partial derivative $\left[\frac{\partial^2 f}{\partial x_i \partial x_j}\right]([B])$ of the function) we can use the following method:

1. Compute *symbolically* the partial derivatives of the function w.r.t. all its variables: $\frac{\partial f}{\partial x_1}(X), ..., \frac{\partial f}{\partial x_n}(X)$.
2. For each x_i, compute the intervals $\left[\frac{\partial^2 f}{\partial x_i \partial x_1}\right]([B])$, ..., $\left[\frac{\partial^2 f}{\partial x_i \partial x_n}\right]([B])$. The intervals can be obtained by differentiating the partial derivative $\frac{\partial f}{\partial x_i}$ with AD in the box $[B]$.

Instead of computing symbolically the partial derivatives of the function, we can build directly a DAG of these derivatives using the backward mode of AD [Iri, 1984]. The elements of the Hessian matrix are computed then performing the backward mode in the graph.

The main advantage of this method is that some subexpressions (internal nodes) shared by several partial derivatives are evaluated only once in the DAG.

2.2.6 Time complexity of interval evaluation and automatic differentiation

The time complexity of the interval evaluation and AD depends on the number n of variables, and on the number e of binary and unary operators of the function f (nodes in the expression tree). A traversal of the expression tree of f is thus $O(e)$.

The time complexity of the natural evaluation of f (i.e., an interval evaluation of the expression tree) is $O(e)$.

The time complexity for computing the interval gradient of f using the symbolic form of the partial derivatives is $O(ne)$ (assuming that the partial derivatives have the same number of binary and unary operators as the function). AD using the forward mode requires $1 + n$ computations in each node of the tree (the evaluation and the gradient operator for each variable). Thus, the complexity of the forward mode is $O(ne)$. AD using the backward mode requires only two computations in each node (evaluation and gradient operator) to compute the gradient of f. Thus, the complexity of the backward mode is time $O(e)$. Observe that the backward mode is the least expensive among the three described methods. The backward method has been used for testing the algorithms described in this thesis.

The time complexity for computing the interval Hessian matrix with AD in the DAG of partial derivatives is $O(ne)$ [Iri, 1984].

2.2.7 Extended interval arithmetic

The definition of the basic interval operators requires the expression be continuous over the input intervals (e.g., $[x]/[y]$ is not defined if $0 \in [y]$). The *extended interval arithmetic* removes this limitation by allowing discontinuities of the expression function (in practice this is the desirable behavior, at least for solving systems of constraints). For example, the expression $\sqrt{[-7,4]}$ produces an error in classical arithmetic. However, using the extended arithmetic, we obtain the image $[0,2]$. The other well-known extended operator is the division:

$$[x]/[y] = (-\infty, +\infty) \text{ if } 0 \in [y]^1$$

When the expression is discontinuous in the input intervals, the image cannot generally be represented by a single interval. As the interval operators are defined as the hull of the image and not as a *union* (the union operator can produce a combinatorial explosion in space and time), it is possible to obtain an overestimation in the image computation (see the third case of non-optimal evaluation in Section 2.3.2). To deal with this problem, some atomic operators are implemented using the union instead of the hull operator (e.g., in [Ratz, 1996], the author has proposed an extension of interval operators aiming at interval Newton-like methods. The `HC4-Revise` narrowing operators, described in Section 3.2.3.4, are also implemented using union).

2.2.8 Intervals and floating point numbers

In practice, when interval-based methods are implemented in computers, the interval bounds are represented with floating point numbers instead of real numbers. This fact forces all the operations with intervals to be outwardly rounded in order to maintain the conservativeness. For example, consider the number $\pi = 3.1415$. If our computer allows only two decimals, π would be represented by the interval $[\pi] = [3.14, 3.15]$, where the left bound of the interval corresponds to a rounding down of 3.1415 to two decimals places, and the right bound of the interval corresponds to a rounding up of 3.1415 to two decimals places. The community concerned with operations with floating point numbers has implemented algorithms for each basic function with the objective of minimizing the number of *lost* floating point numbers. Most of the libraries supporting interval arithmetic have *optimal* arithmetic operators $+,-,/,\times$ (i.e., operators losing less than the distance between two floating point numbers for each bound of the interval image) and more sophisticated algorithms limiting the overestimation in more complex operators like trigonometric functions.

2.3 Properties of interval arithmetic

2.3.1 Conservativeness

We say that f is an **"arithmetic" expression** if it is a composition of binary and unary operators. Any arithmetic expression can be mapped to an interval function replacing the variables of f by interval variables and the operators by interval operators.

One of the fundamental properties of intervals is the *conservativeness*, i.e., the evaluation of the function $[f]$, mapped from an arithmetic expression f, in the box $[B]$ *contains* the image of $[B]$ under f.

[1] The operator is slightly more complicated if the 0 corresponds to a bound of $[y]$.

Proposition 1 (Conservativeness) *Let $f(x_1, ..., x_k)$ be a function and $[f]$ the natural extension of f (i.e., the interval function obtained by mapping f to intervals). Then,*

$$(\forall [B] \in \mathbb{IR}^k) \quad \texttt{Hull}(\mathcal{I}f([B])) \subseteq [f]([B])$$

Proof 1 *Using inductively the definitions of interval operators (2.5) and (2.11) in the syntactic tree of the expression, we can deduce that any vector $X = (x_1, ..., x_k) \in [B]$ satisfies the relation $g(X) \in [g]([B])$ where g represents any subexpression in the tree.* □

Consider Example 10, page 15. For any vector $(x, y, z) \in [x] \times [y] \times [z]$, using (2.5) we deduce that the evaluation of the subexpression $ev_1 = y \times z$ is contained in $[ev_1] = [y] \times [z]$, then using (2.11) we deduce that the evaluation of $ev_2 = (y \times z)^2 = ev_1^2$ is contained in $[ev_2] = ([y] \times [z])^2 = [ev_1]^2$, and so on. Finally, the evaluation of $f(x, y, z)$ is contained in $[f]([x], [y], [z])$.

2.3.2 Non-optimal evaluation

The image of a real function f is generally *not computed optimally* by the evaluation of the mapping of f to intervals. Three important issues make impossible, in the general case, the computation of the optimal image of real functions.

Rounding errors

The rounding errors are caused by the use, in practice, of floating point numbers instead of real numbers (see Section 2.2.8).

Continuity

When a real expression is not continuous over the corresponding intervals, it is possible that its interval evaluation computes an overestimation of the optimal image. Consider for example the expression $(\frac{1}{[x]})^2$, with $[x] = [-1, 1]$ (the expression is discontinuous when $x = 0$). The image of $\frac{1}{[x]}$ is $[-\infty, -1] \cup [1, +\infty]$, and the evaluation computes the hull $[-\infty, +\infty]$. Then, the evaluation of $[-\infty, +\infty]^2$ is $[0, +\infty]$. However, the optimal image is $[1, +\infty]$.

Due to its importance, the third and most important cause for the non-optimality of interval evaluations is discussed in the next section.

2.3.3 Dependency problem

The **dependency problem** is a major obstacle to the application of interval arithmetic. In spite of rounding errors and discontinuities of the expressions, interval methods can compute the image of basic operators very accurately. However, this is not always true with more complicated functions. If a variable occurs several times in a function, each occurrence of it is treated independently by the interval operators. This may lead to an unwanted expansion of the resulting intervals.

Consider for example the function $f(x) = x^2 + x$. The evaluation using interval arithmetic of the interval function $[f]$ (mapped from f) over the interval $[x] = [-1, 1]$ is $[f]([-1, 1]) = [-1, 1]^2 + [-1, 1] = [-1, 2]$,

which is larger than the optimal image of $[x]$: $[-0.25, 2]$. It is possible to find a better expression of f in which the variable x appears only once: $f_1(x) = (x + \frac{1}{2})^2 - \frac{1}{4}$. If we evaluate $[f_1]$, then we reach the optimal image.

Provided that the function is continuous, the optimal image of domains can be achieved if each variable appears only once in the function. However, not every function can be rewritten in this way. The overestimation caused by the dependency problem makes insignificant the other two causes of overestimation (rounding errors and discontinuities of the function). The dependency problem is reduced when the input intervals are smaller, thus justifying the main method based on intervals for solving a system of constraints: the branch & prune (the branching reduces the diameter of intervals limiting the overestimation due to multiple occurrences).

In the next section we describe some algebraic properties of interval arithmetic, especially those that are impacted by the dependency problem.

Properties of interval arithmetic operators

Functions in $\mathbb{IR}$ inherit several algebraic properties from functions in $\mathbb{R}$, such as: Commutativity ($[x] + [y] = [y] + [x]$), associativity ($([x] + [y]) + [z] = [x] + ([y] + [z])$, $([x] \times [y]) \times [z] = [x] \times ([y] \times [z])$), neutral element of the sum ($[x] + [y] = [x] \Leftrightarrow [y] = [0, 0]$) and neutral element of the multiplication ($[x] \times [y] = [x] \Leftrightarrow [y] = [1, 1]$). However, other properties from reals are affected by the dependency problem of intervals. Some of them verify only relaxed properties.

The multiplication is not distributive but it verifies a relaxed property, the *subdistributivity*:

$$[a] \times ([b] + [c]) \subseteq [a] \times [b] + [a] \times [c]$$

The reason is that the developed form is affected by the dependency problem: the interval $[a]$ appears twice in the developed form. For example $[-1, 1] \times ([0, 1] + [-1, 0]) = [-1, 1] \subset [-2, 2] = [-1, 1] \times [0, 1] + [-1, 1] \times [-1, 0]$. The Horner-based extensions (described in Section 2.4.5) are directly related to this property. The objective of these methods is to factorize the common terms of sums implying a sharper evaluation of the new expression.

An example often used for explaining the dependency problem is that generally $[x] - [x] \neq 0$. Only the inclusion is satisfied:

$$0 \in [x] - [x]$$

In the same way $[x]/[x] \neq 1$ but $1 \in [x]/[x]$. In other words, there is no inverse element for the sum and the multiplication (when $\texttt{Diam}([x]) \neq 0$). Furthermore, the subtraction (resp. the division) does not have an algebraic link with the sum (resp. the multiplication). This implies, for example, that the relation $r_1 : [x] = [y] + [z]$ is not equivalent to $r_2 : [x] - [z] = [y]$ because in the algebraic transformation from r_1 to r_2 we used the property $[z] - [z] = 0$ which is false. The correct transformation would be from r_1 to r_2' where $r_2' : [x] - [z] = [y] + [z] - [z]$.

2.4 Interval extensions of real functions

Definition 1 (Interval Extension) *Let f be a real function involving the vector of variables $X =$*

$(x_1, ..., x_k)$. *The interval function* $[f] : \begin{array}{l} \mathbb{IR}^k \to \mathbb{IR} \\ [B] \mapsto [f]([B]) \end{array}$ *is an* **interval extension** *of* f *if:*

$$\begin{array}{ll} (1) & (\forall [B] \in \mathbb{IR}^k) \quad \texttt{Hull}(\mathcal{I}f([B])) \subseteq [f]([B]) \\ (2) & (\forall X \in \mathbb{R}^k) \quad f(X) = [f](X) \end{array}$$

The condition (1) means that for any domain $[B]$ of variables, the image of $[B]$ under the function f is contained in the evaluation of $[f]$. Related to the condition (2), recall that any real value x can be mapped to the interval $[x, x]$. In this case, the input vector of variables X in $[f](X)$ corresponds to a degenerate box, i.e., $X = ([x_1, x_1], ..., [x_k, x_k])$. The condition (2) avoids the definition of pointless extensions like the following one:

$$\begin{array}{l} [f] : \mathbb{IR} \to \mathbb{IR} \\ \quad\; [x] \mapsto [-\infty, +\infty]. \end{array}$$

Example 12 *Consider the function* $f(x) = x^2 - 4x + 4$. *The following interval functions are interval extensions of* f.

$$\begin{array}{ll} [f]_1([x]) = [x]^2 - 4 \times [x] + 4 & [f]_2([x]) = [x] \times ([x] - 4) + 4 \\ [f]_3([x]) = [x] \times [x] + 4 \times (1 - [x]) & [f]_4([x]) = ([x] - 2)^2 \end{array}$$

We can deduce from the example that there exists an infinite number of interval extensions for a given function. Contrarily to real functions, different interval extensions compute different evaluations, due mainly to the dependency problem. We say that an interval extension $[f]$ is *better for the evaluation* than $[g]$ if $[f]$ computes a sharper interval than $[g]$, i.e., $[f]([B]) \subseteq [g]([B])$. In Example 12 using $[x] = [-1, 2]$, we obtain $[f]_1([x]) = [-4, 12]$, $[f]_2([x]) = [-6, 9]$, $[f]_3([x]) = [-6, 12]$ and $[f]_4([x]) = [0, 9]$. The best extension for the evaluation is $[f]_4$.

The **optimal extension** $[f]_{opt}$ of a function f computes the optimal image of any box $[B]$ under f, i.e.

$$(\forall [B] \in \mathbb{IR}^k) \quad [f]_{opt}([B]) = \texttt{Hull}(\mathcal{I}f([B]))$$

In Example 12 an optimal extension of f is given by $[f]_4$. However, in general, the problem of finding the optimal extension is NP-hard [Kreinovich et al., 1997]. For this reason an important goal of interval methods is to compute good and efficient approximations of the optimal image.

Remark that the condition (2) of Definition 1 forces an interval extension to be optimal at least when it is evaluated in a degenerate box. This is a crucial requirement when interval methods are used for solving systems of equations. However conditions (1) and (2) are not enough in practice. We also need that an interval extension gradually decreases the diameter of its evaluation when the diameter of the input decreases.

Definition 2 (Inclusion Monotonic) *Let* $[f]$ *be an interval function.* $[f]$ *is an inclusion monotonic function if*

$$(\forall [B_1] \in \mathbb{IR}^k)(\forall [B_2] \in \mathbb{IR}^k) \quad [B_1] \subseteq [B_2] \Rightarrow [f]([B_1]) \subseteq [f]([B_2])$$

Unary and binary operators are inclusion monotonic. Thus, by induction all the interval functions composed by basic operators are inclusion monotonic too. The next theorem proposed by Moore [Moore, 1966] allows us to prove if an interval function defined by induction corresponds to an interval extension.

Theorem 1 (Fundamental Theorem of Interval Arithmetic) *Let $f(X)$ be a function over the reals and $[f]([B])$ be an interval function. If $[f]$ is inclusion monotonic and $(\forall X \in \mathbb{R}^k)$: $f(X) = [f](X)$ then $[f]$ is an interval extension of f.*

Observe that Theorem 1 is a sufficient condition for interval extensions. Some interval extensions do not fulfill this condition, in particular the famous Taylor extension (see Section 2.4.3).

Now we have the basic tools for computing approximations of domain images under functions. In the next sections we present the main existing interval extensions used for achieving this goal.

2.4.1 The natural extension

The **natural extension** $[f]$ of a real function f corresponds to the mapping of f to intervals.

Proposition 2 (Moore[Moore, 1966]) *Let $[f]$ be the natural extension of $f(x_1, ..., x_k)$. If f is continuous in $[B] = [x_1] \times ... \times [x_k]$, and each variable x_i appears only once in f, then*

$$[f]([B]) = [f]_{opt}([B])$$

However, if a variable occurs more than once in a function f, the interval extension of f generally computes larger intervals than $[f]_{opt}([B])$ does due to the dependency problem. One exception is given by the following property which seems to be new.

Proposition 3 *Let $[f]$ be the natural extension of the continuous and differentiable function $f(x_1, ..., x_k)$. Given a domain $[B]$, if for each variable x_i of f, all the partial derivatives of f w.r.t. the occurrences of x_i have the same sign (e.g., are positive) in each element/point in $[B]$, then*

$$[f]([B]) = [f]_{opt}([B])$$

Proof 2 *Without loss of generality*[1] *consider that the univariate function $f(x)$ is increasing w.r.t. every occurrence of x in the domain $[x]$. Using monotonicity properties (see Section 2.4.2) we can compute the optimal extension:*

$$[f]_{opt}([B]) = \left[\min_{x\in[x]} f(x), \max_{x\in[x]} f(x)\right] = [f(\underline{x}), f(\overline{x})]$$

$g(x_1, ..., x_k)$ is the function obtained from f where each occurrence of x has been replaced by a different variable x_i, $i = 1..k$. As we know, a mapping of a real function to intervals (i.e., the natural extension) treats each occurrence as a different variable (see Section 2.3.3). Then, using Proposition 2: $[f]([x]) = [g]_{opt}([x]^k)$, where $[x]^k = ([x] \times ... \times [x])$.

As g is increasing w.r.t. each variable x_i (because f is increasing w.r.t. each occurrence of x), the minimum of g over $[x]^k$ is $g(\underline{x}, ..., \underline{x}) = f(\underline{x})$ and the maximum is $g(\overline{x}, ..., \overline{x}) = f(\overline{x})$. Then $[f]([x]) = [g]_{opt}([x]^k) = [f(\underline{x}), f(\overline{x})] = [f]_{opt}([B])$. □

Example 13 *Consider the function $f(x, y) = x + x^2y^2 - 3y$. The natural evaluation in the boxes $[B_1] = [0, 1] \times [-1, 1.5]$ and $[B_2] = [1, 2] \times [-5, -3]$ is:*

[1] Abbreviated in the following to W.l.o.g.

$$\begin{array}{lcll} [f]([B_1]) & = & [-4.5, 6.25] & [f]_{opt}([B_1]) = [-4.5, 5] \\ [f]([B_2]) & = & [19, 117] & [f]_{opt}([B_2]) = [19, 117] \end{array}$$

Observe that the natural evaluation in $[B_2]$ is optimal. This is related to Proposition 3: the partial derivative of f w.r.t. each occurrence of x (resp. y) is positive (resp. negative) in every point of $[B_2]$ then the natural extension computes the optimal image.

2.4.2 Monotonicity-based extensions

A particular attention is paid for this interval extension because an important contribution of this thesis is based on monotonicity.

A function f is **monotonic** w.r.t. a variable x in a given domain $[B] = [x_1] \times ... \times [x_k]$ if the evaluation of the partial derivative of f w.r.t. x is positive (or negative) in every point of $[B]$. If the evaluation is positive in every element the function is **increasing** w.r.t. x. On the contrary, if the evaluation is negative in every element then the function is **decreasing** w.r.t. x. For the sake of conciseness, we sometimes write that x is monotonic. A **monotonic function** is a function that is monotonic w.r.t. every variable.

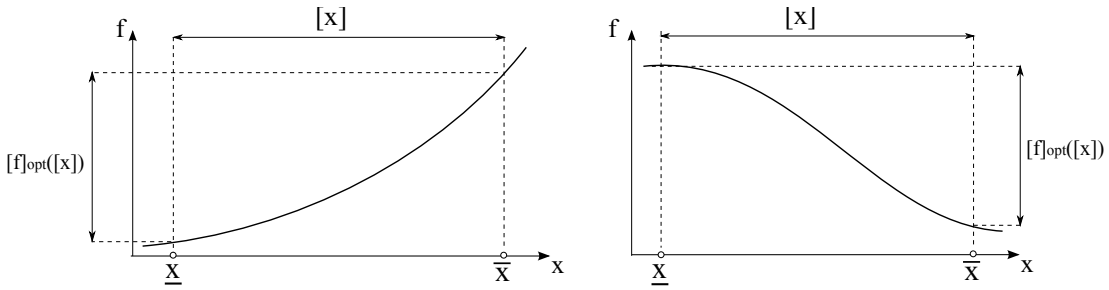

Figure 2.6: A function increasing (left side) and decreasing (right side) w.r.t. x. In both cases the image of $[x]$ can be computed using only the bounds of the interval.

The optimal image of any continuous function $f(x)$ that is monotonic w.r.t. x can be computed using the bounds of the domain of x (see Figure 2.6). Thus, if f is increasing w.r.t. x, the image of $[x]$ under f is:

$$[f]_{opt}([x]) = \left[\min_{x\in[x]} f(x), \max_{x\in[x]} f(x)\right] = [f(\underline{x}), f(\overline{x})] \tag{2.12}$$

symmetrically, if f is decreasing then:

$$[f]_{opt}([x]) = \left[\min_{x\in[x]} f(x), \max_{x\in[x]} f(x)\right] = [f(\overline{x}), f(\underline{x})] \tag{2.13}$$

The unary operators have been defined by using this property in the monotonic parts of the function. For example recall the definition of the square operator:

$$[sqr]([x]) = [x]^2 = \begin{cases} [\underline{x}^2, \overline{x}^2] & \text{if } [x] > 0, \\ [\overline{x}^2, \underline{x}^2] & \text{if } [x] < 0, \\ [0, max(\underline{x}^2, \overline{x}^2)] = [0, |[x]|^2] & \text{if } 0 \in [x]. \end{cases}$$

We can observe three cases. First, when $[x] > 0$ the square function is monotonic increasing w.r.t. x, then the operator is computed using (2.12): $[x]^2 = [\underline{x}^2, \overline{x}^2]$. Second, when $[x] < 0$ the square function is

monotonic decreasing w.r.t. x, then using (2.13): $[x]^2 = [\overline{x}^2, \underline{x}^2]$. Third, if $0 \in [x]$ then the operator is computed dividing the interval into two parts: $[\underline{x}, 0]$ and $[0, \overline{x}]$. The square function is decreasing w.r.t. the first one and increasing w.r.t. the second one, thus $[x]^2 = [0^2, \underline{x}^2] \cup [0^2, \overline{x}^2] = [0, |[x]|^2]$.

The monotonicity property of functions can also be extended to several variables.

Proposition 4 *Let $f(x_1, ..., x_k)$ be a continuous function. If f is monotonic w.r.t. x_i for all $i = 1..k$ in the box $[B] = [x_1] \times ... \times [x_k]$, then the optimal image of the domain under f can be computed:*

$$[f]_{opt}([B]) = [f(x_1^-, ..., x_k^-), f(x_1^+, ..., x_k^+)]$$

where $x_i^- = \underline{x_i}$ and $x_i^+ = \overline{x_i}$ if x_i is increasing. If x_i is decreasing then $x_i^- = \overline{x_i}$ and $x_i^+ = \underline{x_i}$.

Example 14 *Consider the function $f(x, y, z) = -x^2+xy+yz-z$ with domains $[B] = [6, 8]\times[2, 3]\times[7, 15]$. f is increasing w.r.t. y and z, and decreasing w.r.t. x. Then we can compute the optimal image:*

$$[f]_{opt}([x], [y], [z]) = [f(\overline{x}, \underline{y}, \underline{z}), f(\underline{x}, \overline{y}, \overline{z})] = [f(8, 2, 7), f(6, 3, 15)] = [-41, 12]$$

Proposition 4 can be used only in monotonic functions. The *extension by monotonicity* [Hansen and Walster, 2003] computes better evaluations than the natural extension does even when the function is not entirely monotonic, but it is monotonic only w.r.t. some variables.

Definition 3 (Extension by monotonicity) *Let f be a function of variables $X = \{x_1, .., x_k\}$ and domain $[B] = [x_1] \times ... \times [x_k]$. $[B^-]$ (resp. $[B^+]$) is the box obtained from $[B]$ by replacing each interval $[x_i]$, related to an increasing variable, by $\underline{x_i}$ (resp. $\overline{x_i}$) and each interval $[x_j]$ related to a decreasing variable by $\overline{x_j}$ (resp. $\underline{x_j}$). Then, the extension by monotonicity of f in the box $[B]$ is given by:*

$$[f]_m([B]) = \left[\underline{[f]([B^-])}, \overline{[f]([B^+])}\right]$$

where $[f]$ is the natural extension (however, any other extension could be used instead).

For a function f, the extension by monotonicity computes an interval larger than or equal to the optimal extension and sharper than or equal to the natural extension. That is:

$$[f]_{opt}([B]) \subseteq [f]_m([B]) \subseteq [f]([B])$$

The extension by monotonicity *eradicates* the dependency problem related to monotonic variables.

Proposition 5 *Let f be a continuous function in a box $[B]$. If f is monotonic in $[B]$ w.r.t. all its variables appearing several times in the function, then the extension by monotonicity computes the optimal image, i.e.,*

$$[f]_m([B]) = [f]_{opt}([B])$$

Proposition 5 is slightly more general than Proposition 4. The difference lies in that the former allows the presence of *non monotonic* variables in the function. As these variables (appearing once) are optimally evaluated by the natural extension, they do not need to be monotonic.

Example 15 *Consider the intervals* $[x] = [2,3]$, $[y] = [-1,1]$ *and the functions* $f_1(x,y) = x - y^2x$, $f_2(x,y) = xy + 3x - 7y$ *and* $f_3(y) = y^2 - y$. *Knowing that* f_1 *is increasing w.r.t.* x, *and* f_2 *is increasing w.r.t.* x *and decreasing w.r.t.* y, *then we compute the interval extensions by monotonicity:*

$$[f_1]_m([x],[y]) = [[f_1](\underline{x},[y]),[f_1](\overline{x},[y])] = [0,3]$$
$$[f_2]_m([x],[y]) = [[f_2](\underline{x},\overline{y}),[f_2](\overline{x},\underline{y})] = [1,13]$$
$$[f_3]_m([y]) = [[f_3]([y]),[f_3]([y])] = [f_3]([y]) = [-1,2]$$

The extension by monotonicity of f_1 *(resp.* f_2*) is optimal, because* f_1 *(resp.* f_2*) is monotonic w.r.t. all its variables appearing several times, i.e.,* x *(resp.* x *and* y*). The extension by monotonicity of* f_3 *is not optimal and it is equivalent to the natural extension.*

An algorithm for computing the evaluation by monotonicity is more expensive than an algorithm for computing the natural evaluation. It basically consists in three steps. Consider the function $f(X)$, the evaluation $[y] = [f]_m([B])$ is computed by the following algorithm.

1. For each variable $x_i \in X$, the algorithm computes the *interval partial derivatives* $[g_i] = \left[\frac{\partial f(X)}{\partial x_i}\right]([B])$. $0 \notin [g_i]$ implies that f is monotonic w.r.t. the variable x_i in $[B]$. A well-known method for computing interval gradients is the *automatic differentiation* (see Section 2.2.4.2).

2. The boxes $[B^-]$ and $[B^+]$ are generated (initially copied from $[B]$) for computing the left and the right bounds resp. of the interval $[y]$. If f is increasing (resp. decreasing) w.r.t. x_i (i.e., $[g_i] < 0$) the i^{th} interval of $[B^-]$ is replaced by $\underline{x_i}$ (resp. $\overline{x_i}$) and the i^{th} interval of $[B^+]$ is replaced by $\overline{x_i}$ (resp. $\underline{x_i}$). The remaining intervals of $[B^-]$ and $[B^+]$ (i.e., every i^{th} interval such that $0 \in [g_i]$) remains unchanged.

3. The two bounds of $[y]$ are finally computed:

$$\overline{y} = \overline{[f]([B^+])}$$
$$\underline{y} = \underline{[f]([B^-])}$$

Observe that if f is monotonic w.r.t. some variables in $[B^+]$ (resp. $[B^-]$), then we can compute a better approximation of the left bound (resp. the right bound) of the optimal image using the evaluation by monotonicity, that is:

$$\overline{y'} = \overline{[f]([B^{++}])}$$
$$\underline{y'} = \underline{[f]([B^{--}])}$$

where $[B^{++}]$ and $[B^{--}]$ are the boxes generated in the step 2 of the previous algorithm using the monotonicites of f in $[B^+]$ and $[B^-]$ resp. The method can be applied recursively until reaching a fixpoint. This evaluation is implemented in several interval libraries (e.g., `ALIAS` [Merlet, 2000], `Ibex` [Chabert, 2009]). It is formalized as follows.

Definition 4 (Recursive extension by monotonicity) *Let* f *be a function related to the set of variables* $X = \{x_1, ..., x_k\}$ *and domains* $[B] = [x_1] \times ... \times [x_k]$. $[B^-]$ *(resp.* $[B^+]$*) is the box obtained from* $[B]$ *replacing each interval* $[x_i]$ *related to an increasing variable by* $\underline{x_i}$ *(resp.* $\overline{x_i}$*) and each interval* $[x_j]$

related to a decreasing variable by $\overline{x_j}$ *(resp.* $\underline{x_j}$*). Then, the recursive extension by monotonicity of* f *in the box* $[B]$ *is given by:*

$$[f]_{mr}([B]) = \begin{cases} \left[\underline{[f_{mr}]([B^-])}, \overline{[f_{mr}]([B^+])}\right] & \text{if } [B^-] \neq [B] \text{ (or } [B^+] \neq [B]) \\ [f]([B]) & \text{otherwise} \end{cases}$$

Observe that the recursion continues while *new monotonicities* are found. In other words, while the new box is different from the previous one ($[B^-] \neq [B]$ or $[B^+] \neq [B]$).

In the same way as the extension by monotonicity, the recursive extension by monotonicity can be used in combination with other extensions. Consider an interval extension $[f]_x$. With the same assumptions of Definition 4 we can combine the recursive extension by monotonicity with the $[f]_x$ extension as follows:

$$[f]_{mr+x}([B]) = \begin{cases} \left[\underline{[f_{mr}]([B^-])}, \overline{[f_{mr}]([B^+])}\right] & \text{if } [B^-] \neq [B] \text{ (or } [B^+] \neq [B]) \\ [f]_x([B]) & \text{otherwise} \end{cases}$$

In Sections 5.6.4 and 5.6.5 we use two variants of the recursive extension by monotonicity: $[f]_{mr+og}$ using the extension by occurrence grouping ($[f]_{og}$) described in Chapter 5 and $[f]_{mr+h}$ using the Hansen extension ($[f]_h$) described in Section 2.4.4.

If AD (see Section 2.2.4.2) is used for computing the interval gradient, then the time complexity for computing $[f]_m([B])$ is $O(e)$ where e is the number of binary and unary operators in f. The time complexity of $[f]_{mr}$ is $O(ke)$ where k is the number of monotonic variables in the function. The worst case occurs when in each recursion only one new variable is found monotonic.

One of the main contributions of this thesis is the proposition of a new extension based on monotonicity. The idea consists basically in generating new monotonic variables by grouping together only some occurrences of one non-monotonic variable (see Chapter 5).

2.4.3 The Taylor extension

Let f be a continuous and differentiable univariate function in an interval $[x]$. Then,

$$(\forall a \in [x])(\forall b \in [x])(\exists c \in [a,b]): \quad f(b) - f(a) = f'(c)(b-a)$$

This theorem is the so-called *mean value theorem*. Consider the function $h_{a,b}(c) = f(a) + f'(c)(b-a)$. Then $\exists c \in [a,b]$: $f(b) = h_{a,b}(c)$. The optimal extension of $h_{a,b}(c)$ is given by $[h_{a,b}]_{opt}([x]) = f(a) + [f']_{opt}([x])(b-a)$, where $[f']_{opt}$ is the optimal extension of $\frac{df(x)}{dx}$. As $[h_{a,b}]_{opt}$ is inclusion monotonic, then

$$c \subseteq [a,b] \subseteq [x] \Rightarrow [h_{a,b}]_{opt}(c) \subseteq [h_{a,b}]_{opt}([a,b]) \subseteq [h_{a,b}]_{opt}([x])$$

As $b \in [x]$, we obtain finally:

$$(\forall a \in [x])(\forall b \in [x]): \quad f(b) \subseteq f(a) + [f']_{opt}([x])(b-a) \tag{2.14}$$

Proposition 6 *Let* f *be a continuous and differentiable function in the interval* $[x]$ *and* a *be any value of the interval* $[x]$*, then*

$$[f]_{opt}([x]) \subseteq f(a) + [f']_{opt}([x])([x]-a) \subseteq f(a) + [f']([x])([x]-a)$$

where $[f']$ *is an interval extension of* $\frac{df}{dx}$*.*

Proof 3 *The proposition is obtained by extending to intervals the relation (2.14) and using the property:* $[f]_{opt}([x]) \subseteq [f]([x])$. □

The first-order **Taylor extension** in one variable using the midpoint is defined, using Proposition 6, by:

$$[f]_t([x]) = f(\texttt{Mid}([x])) + [f']([x])([x] - \texttt{Mid}([x]))$$

Observe that a has been replaced by the midpoint of the interval.

Figure 2.7 shows an example. The interval function $[f]_t(x) = f(a) + [f']_{opt}([x])(x - a)$ generates a cone containing the curve $f(x)$ for any a in the interval $[x]$. The slopes of the lines bounding the cone

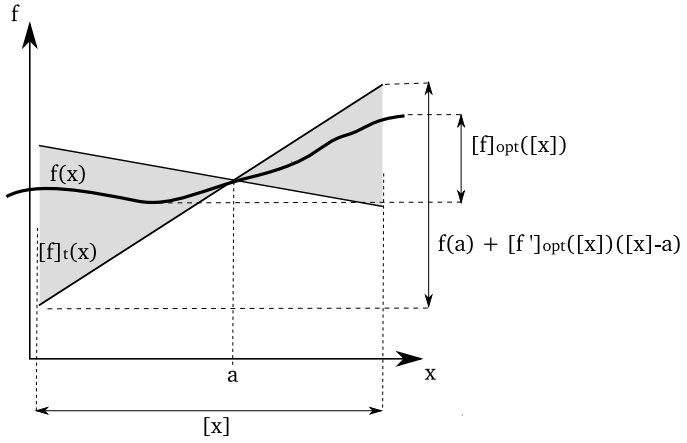

Figure 2.7: The Taylor extension

correspond to the maximum and minimum derivatives of f in the interval $[x]$. The image of $[x]$ under f ($[f]_{opt}([x])$) is contained in the image of $[x]$ under $[f]_t(x)$, i.e., in the interval $f(a) + [f']_{opt}([x])([x] - a)$ corresponding to the Taylor extension of f.

For the general case, we use the mean value theorem extended to several variables (see the proof in [Chabert, 2007], page 26):

Theorem 2 (Mean value theorem) *Let f be a continuous and differentiable function in the box $[B] = \{[x_1], ..., [x_k]\}$. $X = \{x_1, ..., x_k\}$ is the vector of variables related to f. Then,*

$$(\forall A \in [B])(\forall B \in [B])(\exists C \in [B]) : \quad f(B) = f(A) + \sum_{i=1}^{k} \frac{\partial f(C)}{\partial x_i}(b_i - a_i)$$

where a_i and b_i are the i^{th} values of the vectors A and B resp.

Replacing $\frac{\partial f(C)}{\partial x_i}$ by its optimal image for the box $[B]$ ($\left[\frac{\partial f}{\partial x_i}\right]_{opt}([B])$), we obtain:

$$f(B) \subseteq f(A) + \sum_{i=1}^{k} \left[\frac{\partial f}{\partial x_i}\right]_{opt}([B])(b_i - a_i) \tag{2.15}$$

Proposition 7 *Let f be a continuous and differentiable function in the box $[B]$, and A be any value of the box $[B]$, then*

$$[f]_{opt}([B]) \subseteq f(A) + \sum_{i=1}^{k} \left[\frac{\partial f}{\partial x_i}\right]([B])([B] - a_i) \subseteq f(A) + \sum_{i=1}^{k} \left[\frac{\partial f}{\partial x_i}\right]([B])([B] - a_i)$$

Proof 4 *Trivial from (2.15) and using the interval extension property: $[f]_{opt}([B]) \subseteq [f]([B])$.* □

Finally, we deduce the first-order Taylor extension.

Definition 5 (First-order Taylor extension) *Let $f(X)$ be a continuous and differentiable function in the box $[B] = [x_1] \times ... \times [x_k]$ where $X = (x_1, ..., x_k)$ is the corresponding vector of variables of f. The (centered) Taylor extension is defined by:*

$$[f]_t([B]) = f(\texttt{Mid}([B])) + \sum_{i=1}^{k} \left[\frac{\partial f}{\partial x_i}\right]([B])([x_i] - \texttt{Mid}([x_i]))$$

The Taylor extension is not inclusion monotonic (Definition 2), because its evaluation uses an arbitrarily selected value of the interval $[B]$ (the midpoint).

Using the generalization of the mean value theorem it is possible to obtain the Taylor extension for greater orders. For instance, the second-order Taylor extension is defined by:

$$[f]_{t^2}([B]) = [f]_t([B]) + \frac{1}{2}\sum_{j=1}^{k}\Big(([x_j] - \texttt{Mid}([x_j])) \times \sum_{i=1}^{k} \left[\frac{\partial^2 f}{\partial x_i \partial x_j}\right]([B])([x_i] - \texttt{Mid}([x_i]))\Big)$$

Remarks and examples

Consider the function $f_1(x) = 3x^3 - 2x^2 - x$ and the interval $[x] = [-1, 1]$. The derivative of f_1 is $f_1'(x) = 9x^2 - 4x - 1$, and the natural extension yields $[f_1']([-1, 1]) = [-5, 12]$. The function evaluated in the midpoint of $[x]$ is $f(0) = 0$. Then, the Taylor extension computes:

$$[f_1]_t([x]) = 0 + [-5, 12] \times ([-1, 1] - 0) = [-12, 12]$$

The optimum image is $[f_1]_{opt}([-1, 1]) = [-4, 0.098]$ and the natural extension computes $[f_1]([-1, 1]) = [-6, 4]$. In this example, the Taylor evaluation is worse than the natural evaluation. This generally occurs when the interval derivatives have large diameters. Consider now the same function and $[x] = [-0.7, -0.6]$, the Taylor extension computes:

$$[f_1]_t([x]) = f(-0.65) + [f_1'][-0.7, -0.6])([-0.7, -0.6] + 0.65) = [-1.329, -0.7084]$$

The optimum image is $[f_1]_{opt}([-0.7, -0.6]) = [-1.309, -0.768]$ and the natural extension computes $[f_1]([-0.7, -0.6]) = [-1.409, -0.668]$. The Taylor evaluation computes evaluations close to the optimal image when the interval derivative have small diameters. This nice behavior is shared by the extension by monotonicity, because a small interval partial derivative diameter, related for example to a variable x_i, possibly implies that $0 \notin [\frac{\partial f}{\partial x_i}]$ (i.e., f is monotonic w.r.t. x_i). In the last example, f_1 is monotonic w.r.t. x in the interval $[-0.7, -0.6]$, then the evaluation by monotonicity computes the optimal image.

The Taylor extension generally computes sharper images than the extension by monotonicity when the interval partial derivatives have small diameters but contain zero. Consider the function $f_2(x,y) = x^2y - xy^2 - xy$ with intervals $[x] = [-0.2, -0.1]$ and $[y] = [-1.2, -1.1]$. The partial derivatives of f w.r.t. x and y are respectively $g_x(x,y) = 2xy - y^2 - y$ and $g_y(x,y) = x^2 - 2xy - x$. The interval partial derivatives (computed using the natural extension of g_x and g_y) are $[g_x] = [-0.58, 0.47]$ and $[g_y] = [-0.58, 0.48]$. Then the Taylor extension computes:

$$\begin{aligned}[f_2]_t([x]) &= f_2(-0.15, -1.15) + [-0.58, 0.47] \times [-0.15, 0.15] + [-0.58, 0.48] \times [-0.05, 0.05] \\ &= [-0.11025, 0.12175]\end{aligned}$$

that is sharper than the interval computed by the evaluation by monotonicity: $[f]_m([x]) = [f]([x]) = [-0.432, 0.408]$ ($[g_x]$ and $[g_y]$ contain 0 so that the monotonicity extension computes the same interval as the natural evaluation does).

2.4.4 The Hansen extension

Note that the Taylor extension requires that the computation of the interval gradient of the function considers the entire box.

It is possible to use sharper interval partial derivatives than those used by the Taylor extension and Proposition 7. The interval partial derivative related to the i^{th} variable can be computed using a box in which $k - i$ intervals are degenerate. This extension proposed by Hansen and Walster [Hansen and Walster, 2003] allows a better evaluation of functions than the Taylor extension (see the proof in [Hansen and Walster, 2003]).

Definition 6 (Hansen extension) *Let f be a continuous and differentiable function in the box $[B_0] = [x_1] \times ... \times [x_k]$. $X = (x_1, ..., x_k)$ is the vector of variables related to f. $[B_1]$,...,$[B_k]$ are the set of boxes defined by $[B_i] = [x_1] \times ... \times [x_i], \texttt{Mid}([x_{i+1}]) \times ... \times \texttt{Mid}([x_k])$ (a box with $k - i$ degenerate intervals). The (centered) Hansen extension is defined by:*

$$[f]_h = f(\texttt{Mid}([B])) + \sum_{i=1}^{k} \left[\frac{\partial f}{\partial x_i}\right] ([B_i])([x_i] - \texttt{Mid}([x_i]))$$

Remark: Thanks to the automatic differentiation method (see Section 2.2.4.2), the computations of the interval gradient of a function in a box (as the Taylor extension requires) can be done in two traversals of the expression tree. In the case of the Hansen extension, each interval derivative is computed in a different box. If AD is used it would require to traverse twice the expression tree for each variable. The Hansen extension can be implemented as a recursive version of the Taylor extension. The Hansen evaluation is *better* but *more expensive* than the Taylor evaluation.

Just like the Taylor extension, the Hansen extension can also be generalized to higher orders.

2.4.5 Symbolic-based extensions

We design by symbolic-based extension an extension obtained using a symbolic transformation of the original function. The main objective of this process is to find a form for the function that computes sharper evaluations modulo a given extension (e.g., the natural extension, the extension by monotonicity).

Consider for instance the function $f(x,y) = x - xy$ with domains $x = [-1,1]$ and $y = [0,2]$. The natural extension computes $[f]([-1,1],[0,2]) = [-1,1] + [-1,1] \times [0,2] = [-3,3]$. If we factorize the common term x, we obtain the equivalent function $f_1(x,y) = x(1-y)$. The natural extension of f_1 computes the optimal image $[-1,1]$.

Recall that, in case of multiple occurrence of variables, the natural extension does not generally compute an optimal image. For this reason, the factorization of common terms is a well-known method for reducing the dependency problem when the natural extension is used. Reduction in the number of variable occurrences generally implies better natural extensions, mainly thanks to the subdistributivity property:

$$[a] \times ([b] + [c]) \subseteq [a] \times [b] + [a] \times [c]$$

The `ALIAS` library, dedicated to solve systems of equations, can use a `Maple` interface [Merlet, 2000] that allows it to apply symbolic computation techniques (e.g., heuristics using the Horner scheme explained above) for reducing the number of occurrences of a variable in each function of the equation system. Moreover, the symbolic techniques can be applied in any expression that requires being evaluated over intervals[1]. In most of the solvers these expressions are numerically evaluated, i.e., without generating symbolic expressions (e.g., see automatic differentiation in Section 2.2.4.2). The drawback of numerical calculations is that, in general, they do not allow us to reduce the number of occurrences of the evaluated expression which is computed using the natural extension. More details about the symbolic transformations implemented by the `ALIAS-Maple` interface appear in [Merlet, 2000].

Horner-based extensions

For univariate polynomials, the Horner scheme [Horner, 1819] produces the *smallest overall number of arithmetic operations*. Consider the polynomial:

$$f(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{d-1} x^{d-1} + a_d x^d$$

Then the Horner form of $f(x)$ is

$$f_h(x) = (...((a_d x + a_{d-1})x + a_{d-2})x + ...)x + a_0$$

The number d of multiple occurrences of x is the same in $f(x)$ and $f_h(x)$. We cannot thus deduce that $[f_h]([x]) \subseteq [f]([x])$ for univariate polynomials. However, due to the decrease in arithmetic operations, $[f_h]([x])$ is computed faster[2].

Example 16 *Consider the functions $f(x) = x + x^4$, with the Horner form $f_h(x) = x(x^3 + 1)$ and the resp. non-condensed forms $g(x) = x + xxxx$, $g_h(x) = x(xxx + 1)$. Given the interval $[x] = [-2,1]$ the natural extensions of the functions compute:*

$$[f]([x]) = [-2,17] \qquad [f_h]([x]) = [-7,14]$$
$$[g]([x]) = [-10,17] \qquad [g_h]([x]) = [-10,14]$$

[1] For instance, the Jacobian matrix (or partial derivatives) used for computing the Taylor extension (see Section 2.4.3) and required by the interval Newton method (see Section 3.2.1); the projection functions used for enforcing an approximation of the hull-consistency (see Section 3.2.3.2), etc.

[2] In practice, when interval methods are used for solving systems of equations, the impact over the total time depends less on the number of operations of f than on the filtering power.

$[f]([x])$ and $[f_h]([x])$ are not comparable in terms of evaluation. However, the subdistributivity law guarantees the relation $[g_h]([x]) \subseteq [g]([x])$ between the non-condensed forms.

Carrizosa et al. [Carrizosa et al., 2004] propose a Horner-based interval extension for a univariate function f. This extension consists in bisecting the domain of the variable in *zero* and evaluating each subdomain using the *Horner extension* (i.e., the natural extension of the Horner form). This new extension computes an interval *sharper* than or *equal* to the natural and the Horner extensions of f.

In multivariate polynomials, the Horner's scheme can be applied to one variable at a time, considering all the other variables as constants. As we can imagine, the resulting form depends on the order in which the variables are treated. Consider the function $f(x_1, x_2, x_3) = x_1^3x_2 + x_1^2x_3 + x_1^2x_2x_3$. If we apply the Horner's scheme to x_1 we obtain:

$$f_{h(x_1)}(x_1, x_2, x_3) = x_1^2(x_2x_3 + x_3 + x_1x_2)$$

If we now select the variable x_3 we obtain:

$$f_{h(x_1.x_3)}(x_1, x_2, x_3) = x_1^2(x_3(x_2 + 1) + x_1x_2)$$

Observe that, contrarily to univariate polynomials, the Horner's scheme applied to multivariate polynomials reduces the number of occurrences of variables. Consider the interval $[x_1] = [-1, 1]$, $[x_2] = [-8, 1]$ and $[x_3] = [-3, 4]$. The natural extensions of f and $f_{h(x_1.x_3)}$ compute:

$$[f]([x_1], [x_2], [x_3]) = [-43, 36] \quad [f_{h(x_1.x_3)}]([x_1], [x_2], [x_3]) = [-36, 29]$$

Ceberio and Kreinovich propose in [Ceberio and Kreinovich, 2004] two greedy algorithms for selecting the order in which the variables should be treated by the Horner's scheme. The `ALIAS-Maple` interface used by the `ALIAS` library [Merlet, 2000] applies the Horner scheme to expressions using heuristics aiming at reducing the number of variable occurrences.

The nested extension [Ceberio and Granvilliers, 2001; Stahl, 1995]

Consider the *quasi-polynomial* (i.e., a polynomial with expressions as coefficients):

$$f(X) = a_0(X) + a_1(X)m_1(X) + a_2(X)m_2(X) + ... + a_d(X)m_d(X)$$

where X is a vector of variables, $a_i(X)$ is a non-monomial (e.g., sin(x+y)) and $m_i(X)$ is a monomial (e.g., x^2y). The *nested form* is a quasi-polynomial obtained from a sequence of factorizations of two products which are selected if the total degree (the sum of exponents) of their common factors is maximal.

Example 17 *Consider the function $f(x) = 2xy^2z - y^2z^2 + exp(x)x^2z$. First, the nested form factorizes the product y^2z appearing in the first and second term of the sum, i.e.*

$$f_{n(y^2z)}(x) = (2x - z)y^2z + exp(x)x^2z.$$

Finally the nested form is obtained by factorizing the function by z, i.e.,

$$f_{n(y^2z,z)}(x) = ((2x - z)y^2 + exp(x)x^2)z$$

Consider the interval $[x] = [1, 1]$, $[y] = [0, 1]$ and $[z] = [0, 1]$. The natural extension of f computes $[-1, 4.72]$, and the **nested extension** *of f, i.e., the natural evaluation of $f_{n(y^2z,z)}$, computes $[0, 4.72]$.*

More details about the nested form can be found in [Stahl, 1995] and [Ceberio and Granvilliers, 2001].

2.4.6 Combining extensions

The ideal would be to know a priori which extension is the best for evaluating a given expression. This problem is not trivial, because it depends on the form of the expression but also on the domain of variables.

A naive approach is to intersect the intervals computed by the different extensions. However, this approach may result in a lot of useless computations.

A good criterion is to use the known properties of the different extensions. For example, we know that if all the variables appear once in a function (provided the function is continuous over the box), the natural extension computes the optimal image. When the intervals are small, possibly the interval partial derivatives will also be small, then the Taylor extension, the Hansen extension and the extension by monotonicity can be good options. Remark that both the Taylor extension and the extension by monotonicity, need to compute the interval gradient in a box. Then, it seems reasonable to use these extensions together.

Suppose $[y] = [f]_m([B])$, where $[y]$ is the interval obtained by evaluating by monotonicity the function f in the box $[B]$. Recall that the bounds of $[y]$ are computed using the natural extension in the boxes $[B^-]$ and $[B^+]$ with degenerate intervals (see Definition 3 in Section 2.4.2). Instead of the natural evaluation, we can use the Taylor (or Hansen) extension for evaluating the bounds of $[y]$ then a combined extension *monotonicity-Taylor* could be:

$$[f]_{m+t}([B]) = \left[\underline{[f]_t([B^-])}, \overline{[f]_t([B^+])}\right]$$

The same reasoning can be used for combining the recursive extension by monotonicity with the Taylor (or Hansen) extension.

If the only objective of a problem is to compute a good approximation of the domain image under a complicated function, a strategy splitting the variables could be more effective. The box is divided in several subboxes, each subbox is evaluated with some extensions. At the end the different evaluations are hulled. This method is effective (but expensive) for evaluating one function.

Conclusion

In this chapter we have introduced the basic concepts of interval arithmetic: (e.g., basic operators, interval extensions) mainly related with the evaluation (or image computation) of a function in a given domain. In the next chapter we describe several interval-based techniques used for solving systems of constraints.

Chapter 3

Intervals for solving Systems of Equations

Contents

The first motivation to use interval methods for solving systems of equations is the reliability of the computations and the offered control of rounding errors over the floating point numbers. Consider for example the equation $2x + 1/3 = 1$. If we solve it allowing 3 decimal places, possibly we will obtain the solution $x = 0.333$. However, when we will try to verify the result evaluating the left side of the equation, we will obtain $2x + 1/3 = 0.666 + 0.333 = 0.999 \neq 1$. If the computations are performed using intervals we would obtain $x \in [0.333, 0.334]$, that is, the solution of x is certainly in the interval $[0.333, 0.334]$. Evaluating the left side of the equation using interval arithmetic we obtain $2[x]+[1/3] = [0.999, 1.002] \ni 1$.

The second motivation comes from the complete treatment of systems of nonlinear constraints (equations, inequalities) provided by interval methods. A complete treatment means that interval methods are able to find *all the solutions* or to prove that there is no solution in the system. This is the main motivation behind the work presented in this thesis.

Moore [Moore, 1966] proposed the first algorithmic strategy for solving systems of nonlinear equations. This strategy uses a Newton-Raphson operator extended to intervals (described in Section 3.2.1) for converging onto solutions (or to an empty box) when the search space is small enough. Many versions of this operator have been the main concern of the interval analysis community during decades. Moore's strategy also uses a branching operator: the bisection of the domains of variables (box) into two subboxes. The branching allows a complete exploration of the search space with the intent that the Newton-Raphson operator effectively converges onto solutions.

Due to the combinatorial aspect of Moore's method, operators enforcing local consistencies have been added for reducing the search space in polynomial time (local consistency techniques and constraint propagation algorithm are described in Section 3.2.3).

Recently, thanks to the increasing performance of these methods, scientists of different fields in applied mathematics or physics start using interval techniques for solving their problems.

In this chapter we describe the most well-known interval-based strategies and algorithms for solving systems of constraints. Several of them are used by the algorithms proposed in this thesis.

3.1 Solving systems of constraints: the classical interval-based strategy

Using the material introduced in of Chapter 2 we can present a method for finding the set of real solutions of a system of constraints (equations/inequalities) or *numerical CSP*.

Definition 7 *(***Numerical CSP***) A* **numerical CSP** *(NCSP)* $P = (X, C, [B])$ *contains a set of constraints* C *and a set* X *of* n *variables. Every variable* $x_i \in X$ *can take a real value in the interval* $[x_i]$ *and* $[B]$ *is the Cartesian product (called a* **box***)* $[x_1] \times ... \times [x_n]$*. A solution of* P *is an assignment of the variables in* X *satisfying all the constraints in* C*.*

Consider the NCSP $P = (C, X, [B_0])$ related to the set of constraints $C : \{f_1(X) = 0, ..., f_m(X) = 0\}$. The set of solutions of P is given by:

$$S_X = \{X_s, X_s \in [B_0](\forall i = 1..m) : \quad f_i(X_s) = 0\},$$

Consider a precision $\epsilon > 0$. An interval-based strategy finds a set of boxes $S_{[B]}$, with interval diameters less than ϵ, enclosing all the points in S_X (solutions of P), i.e.,

$$X_s \in S_X \Rightarrow \exists [B_s] \in S_{[B]} \text{ such that } X_s \in [B_s]$$

Preferably, a "solution" box contains at least one solution of P.

If it is not properly fixed the ϵ parameter can cause disastrous results. For example in systems with continuous solutions $\epsilon = 10^{-6}$ could return millions of small solutions, decreasing the performance of the solver, while the same value offers good results when the system contains only zero-dimensional solutions.

An algorithm based on the branch & prune method interleaves the following two procedures:

- Evaluation: For each function f_i the algorithm computes an approximation of the image of the current box $[B]$ under f_i (using some interval extension $[f_i]$). If the approximated image does not contain zero ($0 \notin [f_i]([B])$) then the full system has no solution in $[B]$.
- Splitting/Bisection: The interval of one variable in X is split into two parts, generating two new boxes. The procedure (evaluation, bisection) is executed recursively in one box and then in the other. This procedure performs a tree search and carries out a combinatorial solving process.

The recursive process ends (i.e., a leaf of the search tree is reached) when the current box becomes *atomic* (i.e., all the interval diameters of the box are less than ϵ) or when the evaluation proves the non-existence of solution in the current box.

This solving method has very low performance in practice. The actual solving tools replace the bisection/evaluation strategy by a bisection/contraction one, where contraction algorithms reduce domains of variables by eliminating inconsistent values from the bounds of the box with no loss of solutions.

The `Solver` algorithm (Algorithm 1) performs a depth-first search in a tree implemented as a stack. It achieves an exhaustive exploration of the search space with the objective of finding a set of atomic boxes containing all the solutions. The procedure is initialized with a box $[B_0]$ containing the initial domain of each variable in the system. The algorithm ends with two sets of atomic boxes: `certifiedSolutions` and `nonCertifiedSolutions`.

Algorithm 1 `Solver`(**in:** F, X, $[B_0]$, ϵ; **out:** `certifiedSolutions`, `nonCertifiedSolutions`)

```
L ← {[B_0]}
while L ≠ 0 do
  [B] ← L.Pop()
  [B] ← ContractICP([B], ...)
  if [B] ≠ ∅ then
    ([B],certified?) ← ContractNewton([B], ...)
  end if
  if [B] ≠ ∅ then
    if Diam([B]) < ε then
      if certified? then
        certifiedSolutions ← certifiedSolutions ∪ {[B]}
      else
        nonCertifiedSolutions ← nonCertifiedSolutions ∪ {[B]}
      end if
    else
      /* Splitting/bisection */
      ([B_l], [B_r]) ← Split([B], ...)
      L.Push([B_l]); L.Push([B_r])
    end if
  end if
end while
```

The main procedures of the algorithm are:

- `ContractICP`: Filters the box using contractors from (interval) constraint programming. The main objective is to reduce the current box $[B]$ on its bounds with no loss of solutions or, in the ideal case, to obtain an empty box, proving the non-existence of solution. This type of contractors is described in Section 3.2.3.

- `ContractNewton`: Similar to `ContractICP`, the aim of this procedure is the filtering of the box. In addition, if some conditions are satisfied, these operators can determine if the current box $[B]$ contains a unique solution (`certified?` is set to `true`). When this occurs, some iterations of `ContractNewton` quickly converge to an atomic box containing the *certified solution*. These operators are described in Section 3.2.1.

- `Split`: When the current box becomes neither an atomic box nor an empty box, the interval of one variable is split into two parts (bisection), which generates two new boxes $[B_l]$ and $[B_r]$ that are pushed into L. Observe that the management of L as a stack implements a depth-first search. More clever splitting techniques for selecting the next variable interval to be split are described in Section 3.3.

3.2 Filtering/contraction algorithms

Filtering/contraction consists in eliminating values from the domains of variables (search space) that do not satisfy one or more constraints in the system. Consider the function $f(x, y) = x^2 + y$, with domains $[x] = [-1, 2]$ and $[y] = [-1, 0]$. The equation $f(x, y) = 0$ is inconsistent when x is greater than 1. Indeed, if $x > 1$ then there does not exist any value $y \in [y]$ such that $f(x, y) = 0$. Then, a filtering technique could reduce the domain of x to $[-1, 1]$. Remark that filtering techniques do not lose any solution of the system.

Definition 8 (Filtering) *Let $P = (C, X, [B_0])$ be an NCSP, where C is the set of constraints and X is the set of variables. We call* **filtering** *(or* **contractor***) a function Φ_P that accepts as input a box $[B] \subseteq [B_0]$ and returns a box $\Phi_P([B])$ such that:*

If $X_s \in [B]$ is a solution of P then $X_s \in \Phi_P([B])$	*(conservative)*
$\Phi_P([B]) \subseteq [B]$	*(contractive)*
If $[B'] \subseteq [B]$ then $\Phi_P([B']) \subseteq \Phi_P([B])$	*(monotonic)*

In this thesis, we distinguish three kinds of contraction algorithms:

- Contractors from interval analysis.
- Contractors from constraint programming, including constraint propagation algorithms and algorithms achieving stronger partial consistencies.
- Linear relaxation algorithms.

In the next sections, we detail these contractors with a focus on the constraint propagation algorithms.

3.2.1 Operators from interval analysis

One of the most effective filtering algorithms used in interval analysis is the extension of the Newton method to intervals (*interval* Newton). The objective of the classical Newton method is to find successively better approximations to the zeros of a real function. Consider a continuous and differentiable function $f(x)$ and its derivative $f'(x)$. If $x^{(l)}$ is an approximation of a solution of the equation $f(x) = 0$, then a better approximation is obtained by:

$$x^{(l+1)} = x^{(l)} - \frac{f(x^{(l)})}{f'(x^{(l)})}$$

The interval Newton method generalizes the procedure to intervals. Consider a function $f(x)$ and an initial interval $[x]$. The procedure applies successively the contraction step $[x^{(l+1)}] \leftarrow [x^{(l)}] \cap N([x^{(l)}])$ for contracting the interval (l indicates the iteration number), where N is the *Newton operator*. If the iterations converge to a fixpoint, then the procedure will return an atomic interval containing the only solution of the equation $f(x) = 0$ in $[x]$. If an iteration returns an empty interval, then there is no solution in $[x]$. When the iterations do not converge to a fixpoint, we cannot predict the presence or absence of solutions in $[x]$.

For deducing the Newton operator, we use the mean value theorem:

$$(\forall x \in [x])(\forall x_m \in [x])(\exists c \in [x]) : \quad f(x) = f(x_m) + f'(c) \times (x - x_m) \tag{3.1}$$

where f' is the derivative of f. As $f(x) = 0$ we can isolate the variable x from (3.1) as follows:

$$f(x) = 0 \quad \Rightarrow \quad (\forall x_m \in [x])(\forall x \in [x])(\exists c \in [x]) : \quad 0 = f(x_m) + f'(c) \times (x - x_m) \tag{3.2}$$

$$f(x) = 0 \quad \Rightarrow \quad (\forall x_m \in [x])(\forall x \in [x])(\exists c \in [x]) : \quad x = x_m - \frac{f(x_m)}{f'(c)} \tag{3.3}$$

Finally, the *Newton contractor* is obtained by an interval extension (e.g., the natural extension) of the right side of the equation (3.3):

$$N([x]) = x_m - \frac{f(x_m)}{[f']([x])} \tag{3.4}$$

where $[f']$ is an interval extension (e.g., the natural extension) of f' and $x_m = \texttt{Mid}([x])$ (or any other point chosen from the interval $[x]$).

The method explained above is called **univariate interval Newton**. Due to the discontinuity of the division operator (when $0 \in [f']([x])$) it is possible that the natural evaluation of $N([x])$ is overestimated. To avoid this overestimation, it is necessary to use an extended version of the interval division to return a union instead of the hull of the intervals (for more details see the work of Ratz [Ratz, 1996]). The intersection and subtraction should also be modified accordingly.

Example 18 *Consider the function* $f(x) = x^2 - 4$ *with the initial domain* $[x] = [1, 5]$. *The first three iterations of the Newton method compute:*

$$[x] \leftarrow [x] \cap N([x]) = [1,5] \cap 3 - \tfrac{f(3)}{[f']([1,5])} = [1, 2.5]$$
$$[x] \leftarrow [x] \cap N([x]) = [1,2.5] \cap 1.75 - \tfrac{f(1.75)}{[f']([1,2.5])} = [1.9375, 2.21875]$$
$$[x] \leftarrow [x] \cap N([x]) = [1.9375,2.21875] \cap 2.07813 - \tfrac{f(2.07813)}{[f']([1.9375,2.21875])} = [1.9959, 2.00633]$$

The generalization of the method to n variables and n equations is obtained by using the mean value theorem (see Section 2). It consists in replacing in (3.2) the variable x by a vector of variables $X = \{x_1, ..., x_n\}$, f by a vector of functions $F = \{f_1, ..., f_n\}$ and the derivative of f by the Jacobian matrix. This matrix contains in each cell (i, j) the partial derivative of f_i w.r.t. the variable x_j ($\frac{\partial f_i}{\partial x_j}$).

$$F(X) = 0 \Rightarrow (\forall X_m \in [B])(\exists \mathbf{A} \in [\mathbf{A}]) : \quad 0 = F(X_m) + \mathbf{A}.(X - X_m) \tag{3.5}$$

where $[\mathbf{A}]$ is an interval evaluation of the Jacobian matrix in the box $[B]$.

The relation can also be written as follows:

$$F(X) = 0 \Rightarrow (\exists \mathbf{A} \in [\mathbf{A}]) : \quad \mathbf{A}.Y = -F(\texttt{Mid}([B])) \tag{3.6}$$

where X_m has been replaced by $\texttt{Mid}([B])$ and $Y = X - \texttt{Mid}([B])$. In sections 3.2.1.1 and 3.2.1.2, we describe classical techniques for finding a box $[Y_s] \subseteq [B]$ that contains all the solutions that satisfy the linear system in the right side of the relation 3.6, i.e.,

$$[Y_s] \supseteq \{Y, Y \in ([B] - \texttt{Mid}([B])), (\exists \mathbf{A} \in [\mathbf{A}]), \mathbf{A}.Y = -F(\texttt{Mid}([B]))\} \tag{3.7}$$

As $X = \texttt{Mid}([B]) + Y$, the *multivariate interval Newton operator* is obtained extending to intervals the left side of the equation:

$$N([B]) = \texttt{Mid}([B]) + [Y_s] \tag{3.8}$$

Example 19 *Consider the two equations* $f_1(x_1, x_2) = x_1^2 + x_2^2 - 1 = 0$ *and* $f_2(x_1, x_2) = (x_1 - 1)^2 + x_2^2 - 1 = 0$ *and the initial intervals* $[x_1] = [0.2, 0.8]$ *and* $[x_2] = [0.6, 0.9]$ *(*$[B^{(0)}] = \{[x_1], [x_2]\}$*).*

First we find a box $[Y_s]$ *that satisfies the relation (3.7), where* $[\mathbf{A}] = \begin{pmatrix} [0.4, 1.6] & [1.2, 1.8] \\ [-1.6, -0.4] & [1.2, 1.8] \end{pmatrix}$ *is the interval Jacobian matrix evaluated in* $[B^{(0)}]$ *(from [Bliek, 1992]).*

The interval Newton operator uses the box $[Y_s]$ *to obtain the first domain reduction (*$[B^{(1)}] = Mid([B^{(0)}]) + [Y_s]$*):*

$$[B^{(1)}] = \{[0.406, 0.594], [0.854, 0.9]\}$$

The following iterations make the box converge quickly to the unique solution of the system in $[B^{(0)}]$*:*

$$[B^{(2)}] = \{[0.499, 0.501], [0.8658, 0.8664]\}$$
$$[B^{(3)}] = \{[0.4999999, 0.5000001], [0.86602538, 0.86602544]\}$$

There exist many variants of the interval Newton method differing in:

1. The used matrix (instead of the Jacobian matrix).
2. The use of symbolic manipulation of the matrix coefficients with the objective of decreasing the number of multiple occurrences of variables [Merlet, 2002].
3. The way the system is linearized. For instance, Krawczyk (see Section 3.2.1.3), Borsuc, Kantorovich (see Section 3.2.1.4), etc.
4. The technique used for solving the linear system: matrix inversion, Gauss-Seidel, Hansen-Bliek [Hansen and Walster, 2003], Gauss elimination, LU, etc.
5. The use of preconditioning for obtaining more effective contractions.

For instance, the interval Newton contractor implemented in `Ibex` [Chabert, 2009; Chabert and Jaulin, 2009c] (the interval library used for our experiments) uses the preconditioned Hansen matrix and solves the linear system using a Gauss-Seidel procedure.

The main strength of interval Newton contractors is that it considers all the system as one only global constraint reducing all the variables at the same time. The main drawback is that it only works with square *well-constrained* systems (i.e., square systems of independent equations with a finite number of solutions) only converging when the box is small enough and contains a unique solution. Local contractors (contractors that consider one constraint at a time) from interval constraint programming seem to be a good alternative when the Newton iteration is not effective (see Section 3.2.3).

In the next section, we describe the Gauss-Seidel method. This technique is often used in solvers for solving the linear system (3.7). Section 3.2.1.2 describes the preconditioning, a technique commonly used for improving the performance of the method (e.g., Gauss-Seidel) used for solving the linear system (3.7).

3.2.1.1 Gauss-Seidel method

The Gauss-Seidel method is a well-known iterative method for solving linear systems of equations. The extension of the method to intervals is commonly used by the interval Newton procedure for finding a set of values $[Y_s] \subseteq [Y]$ satisfying the relation (3.7), i.e.,

$$[Y_s] \supseteq \{Y \in [Y], (\exists \mathbf{A} \in [\mathbf{A}]), \mathbf{A}.Y = D\}$$

where $[\mathbf{A}]$ is a square interval matrix and D is a vector. The Gauss-Seidel method applies the iteration $[Y] \leftarrow [GS]([Y])$ until reaching the fixpoint converging to the solution. $[GS]$ is the **Gauss-Seidel operator**. Each element of the vector $[GS]([Y]) = ([GS_1]([y_1]), ...[GS_m]([y_m]))^T$ is given by:

$$[GS_i]([y_i]) = -\frac{\sum_{i \neq j} [a_{ij}][y_j] - d_i}{[a_{ii}]},$$

where d_i is the i^{th} element of the vector D and $[a_{ij}]$ is the value of the cell (i, j) of the interval matrix $[\mathbf{A}]$.

3.2.1.2 Preconditioning

The preconditioning is a technique that transforms the problem of finding the set

$$\{Y \in [Y], (\exists \mathbf{A} \in [\mathbf{A}]), \mathbf{A}.Y = D\}$$

into finding the set

$$\{Y \in [Y], (\exists \mathbf{A}' \in \mathbf{P}.[\mathbf{A}]), \mathbf{A}'.Y = P.D\}$$

by performing a left multiplication of $[\mathbf{A}]$ and D by a punctual matrix $\mathbf{P}$ such that the new problem has a better "numerical behavior" and contains the set of solutions of the original one.

The preconditioning is commonly used in the interval Newton method for improving the precision of the solution $[Y_s]$ (see the relation (3.7)) when it is solved by some solving technique (e.g., Gauss-Seidel).

Example 20 *Consider the well-known ill-conditioned Hilbert matrix. Each cell a_{ij} in the Hilbert matrix is equal to $\frac{1}{i+j-1}$. Then the 3×3 (approximated) Hilbert matrix is:*

$$\mathbf{A} = \begin{pmatrix} 1 & 0.5 & 0.333 \\ 0.5 & 0.333 & 0.25 \\ 0.3333 & 0.25 & 0.2 \end{pmatrix}$$

If we solve the equation $\mathbf{A}.Y = D$ using a classic method, with $D = (1, 1, 1)^T$ we obtain:

$$Y_s = (3.04, -24.21, 30.19)$$

Now, if we add a small uncertainty: $D = (1.1, 1.1, 1.1)^T$ the new result amplifies the uncertainty:

$$Y_s = (3.95, -27.84, 33.22)$$

In intervals methods, the ill-conditioning implies large intervals producing a big overestimation in the resolution of the interval equation.

The most used preconditioning consists in using the matrix $\mathbf{P} = \texttt{Mid}([\mathbf{A}])^{-1}$, where $\texttt{Mid}([\mathbf{A}])$ is a real matrix obtained by replacing each interval element of the interval matrix $[\mathbf{A}]$ by its midpoint. This preconditioning implies that each interval $[a'_{ii}]$ on the diagonal of $\mathbf{P}.[\mathbf{A}]$ is large and the intervals $[a'_{ij}]$ $(j \neq i)$ are small. Consider that $[a'_{ij}]$ represents an element of the preconditioned matrix $(\mathbf{P}.[\mathbf{A}])$, $[y_i]$ represents an element of the vector $[Y]$ and d'_i represents an element of the vector $\mathbf{P}.D$. If the Gauss-Seidel method is used for solving the system, the following iteration is applied in each equation i (the procedure iterates over all the equations until reaching the fixpoint):

$$[y_i] \leftarrow [y_i] \cap -\frac{[a'_{1i}][y_1] + ... + [a'_{(i-1)i}][y_{i-1}] + [a'_{(i+1)i}][y_{i+1}] + ... + [a'_{ni}][y_n] + d'_i}{[a'_{ii}]}$$

As the intervals in the numerator of the fraction are small, the operator probably will perform a good contraction over the interval $[y_i]$.

The preconditioning performs a kind of reparation of an ill-conditioned matrix in order to obtain a more precise solution of the related linear system.

3.2.1.3 The Krawczyk operator

The Krawczyk operator ([Krawczyk, 1969]) can be used as an alternative of the Newton operator defined by (3.8), page 39. Thus, the contraction step using the Krawczyk operator is given by:

$$[B] \leftarrow [B] \cap K([B])$$

where K is the Krawczyk operator.

In the same way as the Newton operator, the Krawczyk operator is deduced from the extension of the mean value theorem to intervals, i.e.:

$$(\forall X_m \in [B])(\forall X \in [B])(\exists \mathbf{M} \in [\mathbf{A}] : \quad F(X) - F(X_m) = \mathbf{M}(X - X_m)$$

where $[\mathbf{A}]$ is the interval Jacobian matrix of the function F in the box $[B]$. However, instead of using the mean value theorem to obtain a linear system with interval coefficients, $\mathbf{M}$ is decomposed into $(\mathbf{M}-\mathbf{I})+\mathbf{I}$. This decomposition still allows us to isolate X:

$$F(X) = 0 \Rightarrow (\forall X_m \in [B])(\exists \mathbf{M} \in [\mathbf{A}]) : \quad X = X_m - F(X_m) + (I - \mathbf{M})(X - X_m)$$

Thus, extending the result to intervals, the Krawczyk operator is defined by:

$$K([B]) = X_m - F(X_m) + (\mathbf{I} - [\mathbf{A}])([B] - X_m)$$

where $X_m = \texttt{Mid}([B])$ (or any other point arbitrarily chosen from the interval $[B]$).

Like for the Newton operator, the interval Jacobian matrix can be preconditioned as shown in Section 3.2.1.2.

3.2.1.4 Kantorovich's theorem

Kantorovich's theorem (described in [Demidovitch and Maron, 1973; Ortega and Rheinboldt, 1970; Tapia, 1971]), using the second derivatives of the function (the Hessian matrix), allows us to construct, under certain conditions imposed by the theorem, a box $[B']$ such that $X_0 \in [B'] \subset [B]$, where X_0 is the starting point of the method.

If $[B']$ is successfully generated, then two interesting properties are verified:

- There exists a *unique* solution in $[B']$.
- A classical interval Newton method starting with X_0 is able to converge to the solution.

Kantorovich's theorem can be used to prove the existence and uniqueness of solutions in a box. Moreover, the theorem can be used to eliminate boxes that do not contain any solution or to filter boxes that are close to a certified solution ([Merlet, 2000]).

3.2.2 Linear relaxation

Linear relaxation is often used in *global optimization* problems that consist in minimizing an objective function under nonlinear constraints. Linear relaxation based methods are commonly used to speed up the convergence to a global optimum.

The principle consists in approximating the nonlinear constraints by a set of enclosing linear inequalities generally treated by a Simplex algorithm. We can make three important remarks:

- The enclosing linear system explains the term relaxation. Indeed, the linear system contains more solutions than the nonlinear system. In case of optimization, relaxing the objective function allows us to find a lower bound of the optimal value.

- If the linearization is performed without taking into account some considerations due to floating point numbers, it is possible, in pathological cases, to miss the conservativeness property of intervals, thus resulting in a loss of solutions. In this respect `QuadSolver` [Lebbah et al., 2005] and `GlobSol` [Kearfott et al., 1996] are *rigorous* (i.e., they do not lose solutions), contrarily to `Baron` [Sahinidis and Twarmalani, 2002].

- Most of the available Simplex algorithms handle floating point numbers and are not rigorous. For obtaining reliable upper bounds for the optimum, `QuadSolver` implements, for example, a cheap postprocessing procedure introduced in [Neumaier and Shcherbina, 2004], based on *directed roundings* and interval arithmetic.

Linear relaxation is used in several tools like `Globsol`, `QuadSolver` or `Baron`.

In the case of solving system of constraints, the contraction algorithm `Quad` [Lebbah et al., 2005] combines linear relaxation and the algorithm `HC4` (detailed in Section 3.2.3.4) of constraint programming. The basic steps of the `Quad` contractor are:

1. The system is reformulated: the nonlinear terms are replaced by a new variable (e.g., x^2 by y_i).

2. The new system is extended by introducing redundant linear constraints to provide tight linear approximations of the nonlinear terms. Then, a Simplex algorithm is used to reduce the bounds of each variable x (i.e., $\underline{x} = min\ x$ in LP and $\overline{x} = max\ x$ in LP, where LP corresponds to the subsystem of linear constraints).

3. The reductions performed in step 2 are propagated to the whole system (including the nonlinear terms) by using the `HC4` algorithm.

The main differences among the algorithms in this community come from the way the linear relaxation is obtained. For an exhaustive description of the existing techniques see [Sherali and Adams, 1999].

Due to the success of these approaches in combinatorial global optimization, it seems important to make a comparison with the methods of interval analysis and with interval constraint programming. Arnold Neumaier and the COCONUT project have contributed to integrate the `2B` algorithm inside `Baron`. The algorithm `CIRD` [Vu et al., 2009a], the global optimization tool by Lebbah, Rueher, Michel, Goldsztejn [Lebbah et al., 2005; Rueher et al., 2008] and the tool by Baharev and Ré [Baharev and Rév, 2009] are first examples integrating linear relaxation with constraint programming algorithms.

3.2.3 Constraint propagation algorithms

The constraint propagation algorithms are issued from constraint programming over intervals. These algorithms focus on the domain reduction w.r.t. *only one* equation/constraint. A propagation algorithm, similar to the well-known `AC3` algorithm in finite domain CSPs, is used for propagating the reductions to all the system until reaching a fixpoint. Algorithm 2 describes the basic procedure. $var(c)$ corresponds to the set of variables implied in the constraint c.

Algorithm 2 `Propagation`(**in:** C, X, τ_{propag}); **out:** [B]

$Q \leftarrow \{c, c \in C\}$
while $Q \neq \emptyset$ **do**
  $c \leftarrow Q$.`Pop`()
  $[B'] \leftarrow$ `Revise`$(c, X, [B])$
  /* $[B] = \{[x_1], ..., [x_k]\}$ and $[B'] = \{[x'_1], ..., [x'_k]\}$ */
  for all $x_i \in var(c)$ such that $\frac{\texttt{Diam}([x_i]) - \texttt{Diam}([x'_i])}{\texttt{Diam}([x_i])} > \tau_{propag}$ **do**
    /* Constraint propagation */
    for all $c' \in C$ such that $x_i \in var(c')$ **and** $c' \neq c$ **do**
      Q.`Push`(c')
    end for
  end for
  $[B] \leftarrow [B']$
end while

At first, the *propagation queue* Q is initialized with all the constraints of the system. Then, an iteration handles each constraint in Q until the queue becomes empty. The `Revise` procedure performs the contraction, i.e., it eliminates inconsistent values from the bounds of the box that do not satisfy the constraint c.

After that, each variable x_i implied in c with a reduction ratio on the domain size superior to τ_{propag} propagates the changes to the other constraints involving x_i (in the code, every c' involving x_i is pushed into Q).

Fixpoint of constraint propagation

The fixpoint of the `Propagation` algorithm is reached when no variable is reduced more than τ_{propag} with the current `Revise` procedure.

τ_{propag} is a precision parameter used for avoiding a slow convergence of the propagation. In Algorithm 2 τ_{propag} corresponds to a relative size of the interval. It is also commonly implemented as a fixed interval size. In `Ibex` the default value of the τ_{propag} parameter used in the propagation algorithm `HC4` is fixed to 10%.

The final contraction depends on the order in which the constraints are treated by the propagation. Due to the fact that computers work with floating point numbers, τ_{propag} *must be* greater than or equal to the diameter of two consecutive floating point numbers. When τ_{propag} is equal to this diameter, the filtering performed by the propagation algorithm is unique even when the order between the constraints changes.

The propagation algorithms used by interval-based methods all use a similar `AC3`-like propagation procedure. They mainly differ in the `Revise` procedure.

In the next sections we review the main `Revise` methods enforcing partial consistencies with the objective of reducing the domains of variables. We will start with the well-known *Arc-consistency* issued from finite domain CSPs.

3.2.3.1 Arc-consistency

Definition 9 (Arc-consistency for numerical CSPs) *Let c be a constraint involving the vector of variables $X = (x, y_1, ..., y_k)$. $[B] = [x] \times [y_1] \times ... \times [y_k]$ is the domain of X. The pair (c, x) is* **arc-consistent** *if for every value $v \in [x]$ there exists a vector $(v_1, ..., v_{i-1}, v_{i+1}, ..., v_n) \in [y_1] \times ... \times [y_k]$ such that $c(v, v_1, ..., v_n)$ is satisfied.*

An NCSP $P = (C, X, [B])$ is arc-consistent if each pair (c, x) with $c \in C$ and $x \in X$ is arc-consistent.

In general, enforcing arc-consistency in a constraint system can lead to a combinatorial explosion [Hyvönen, 1992].

Example 21 *Consider the constraint $c_1 : sin(x) = y$ with domains $[x] = [0, 31416]$ and $[y] = [0.5, 0.7]$. Due to the periodicity of the sine function, enforcing arc-consistency leads to the union of 10^4 intervals for the variable x.*

Partial consistencies arising from finite domain CSPs guarantee conditions over all the elements of a domain. This does not seem reasonable when we lead with constraints on the real numbers (actually on the floating point numbers) due to the very large domains. Thus, a more reasonable approach is to guarantee conditions only over the bounds of the domain avoiding combinatorial explosion. This brings the community to the definition of new partial consistencies.

3.2.3.2 Hull-consistency

Consider the constraint $c : f(X) = 0$. Close to the idea of computing the exact/optimal range of an interval function f, the objective of a revise procedure is the optimal *filtering* of $[B]$, i.e., the elimination of *all* the values, from the bounds of $[B]$, that do not satisfy c. The elimination of inconsistent values from the domain of a variable is called *projection*. The smallest box containing all the solutions of c in $[B]$ is called *hull-consistent* or 2B-consistent [Lhomme, 1993]. The hull-consistency is close to the *bound-consistency* used in finite domain CSPs [Van Hentenryck et al., 1994].

Definition 10 (Projection of a constraint) *Let c be a constraint involving the variables $\{x_1, ..., x_k\}$. and $[B] = [x_1] \times ... \times [x_k]$ be the corresponding domains. The projection of c over x_i restricted to the box $[B]$ (denoted $\Pi_x^c([B])$) is:*

$$\Pi_x^c([B]) = \{v_i \in [x_i], \exists (v_1, ..., v_{i-1}, v_{i+1}, ..., v_k) \in [B] \text{ such that } c(v_1, ..., v_k) \text{ is satisfied}\}$$

Note that the projection of a constraint over a variable does not necessarily yield an interval. When the function is discontinuous or non-monotonic over the domain, the corresponding projections may correspond to a set of intervals. For simplicity we sometimes call projection the hull of this set of intervals, although the correct term should be **projection hull**.

Example 22 *Consider the constraint* $c : x^2 = y$ *with domains* $[x] = [-2, 2]$ *and* $[y] = [1, 4]$*, the projection of* c *over* x *is* $\Pi_x^c([B]) = [-2, -1] \cup [1, 2]$*. The projection hull is* $\texttt{Hull}(\Pi_x^c([B])) = [-2, 2]$*.*

Definition 11 (Hull-consistency) *Let* c *be a constraint involving the vector of variables* $X = (x, y_1, ..., y_k)$*.* $[B] = [x] \times [y_1] \times ... \times [y_k]$ *is the domain of* X*. The pair* (c, x) *is hull-consistent in* $[B]$ *if:*

$$\exists v_1, ..., v_k \in [y_1] \times ... \times [y_k] \text{ such that } c(\underline{x}, v_1, ..., v_k) \text{ is satisfied,}$$
$$\exists v_1, ..., v_k \in [y_1] \times ... \times [y_k] \text{ such that } c(\overline{x}, v_1, ..., v_k) \text{ is satisfied.}$$

A constraint c *is hull-consistent if for all* $x \in var(c)$*, (c,x) is hull-consistent. An NCSP* $P = (C, X, [B])$ *is hull-consistent if each pair* (c, x) *with* $c \in C$ *and* $x \in X$ *is hull-consistent in* $[B]$*.*

The hull-consistency is restricted to the bounds of the variable domains, but some values inside the domain may be inconsistent.

Projections over the variables of a constraint are **idempotent**, and the results are the same regardless of the order in which they are applied. Then, for enforcing the hull-consistency of a constraint c, it is enough to project once over each variable involved in c. In practice however, projections are often computed using a precision (ϵ). Thus, the results could depend on the order in which they are applied.

3.2.3.3 Hull-consistency of primitive constraints

There exist several difficulties for computing *optimal* projection hulls (thus for enforcing hull-consistency) in polynomial time. The reasons are the same as for obtaining an optimal evaluation, i.e., rounding errors, discontinuity of the function and the dependency problem (see Section 2.3.2).

Consider the constraint $c : x + y = z$ with $x = [-1, 1]$, $[y] = [1, 2]$ $[z] = [1, 1]$. The more intuitive method to compute the projection of c over x consists in isolating the variable and computing the projection using interval arithmetic, i.e., intersecting the current interval $[x]$ with an interval evaluation of the expression $z - y$, i.e.,

$$[x] \leftarrow [x] \cap ([z] - [y]) = [-1, 1] \cap ([1, 1] - [1, 2]) = [-1, 0]$$

In this simple example the projection is optimal. However, if z or y appears several times, we will face with the dependency problem in the projection evaluation. If the variable x appears several times in c, the isolation will become difficult or impossible.

As variables with multiple occurrences cannot generally be isolated, the method adds the *projection functions* corresponding to the isolation of each occurrence of variables. This method enforces the *hull-consistency of the primitive constraints* related to a constraint [Collavizza et al., 1999]. The primitive constraints of a constraint c are obtained by decomposing c into a set of binary or ternary constraints semantically equivalent to c.

Remark

The `ALIAS` framework, implemented by Jean-Pierre Merlet [Merlet, 2002], improves the filtering power of each projection function in `Maple` by using symbolic computations (for example reducing the multiple occurrences of variables using the Horner form, see Section 2.4.5). Moreover, if the option allowing the use of derivatives is activated, the projection functions are evaluated by monotonicity (see Section 2.4.2).

Example 23 *Consider the constraint* $c : x^2 - x + y = 0$ *with domains* $[x] = [1, 8]$ *and* $[y] = [-1, 1]$. *For computing the approximate projection of* c *over* x, *we generate two projection functions isolating the occurrences of* x*:*

$$p_1(x, y) = \sqrt{x - y}$$
$$p_2(x, y) = x^2 - y$$

The functions are then evaluated using the natural extension, i.e., $[x] \leftarrow [x] \cap [p_1]([x], [y]) = [1, 8] \cap \sqrt{[1, 8] - [-1, 1]} = [1, 3]$ *and* $[x] \leftarrow [x] \cap [p_2]([x], [y]) = [1, 3] \cap ([1, 3]^2 - [-1, 1]) = [1, 3]$. *The approximate projection obtained is* $[1, 3]$.

`HC3-Revise` [Benhamou et al., 1994] and `HC4-Revise` [Benhamou et al., 1999] are cheaper methods for computing the hull-consistency of primitive constraints.

`HC3-Revise` decomposes the constraint c into a set of primitive constraints. The primitive constraints are projected over their variables using arithmetic evaluations and *narrowing operators*. `HC4-Revise` performs a kind of automatic projection (similar to the automatic differentiation method presented in Section 2.2.4.2) with no need of decomposing the constraint.

Narrowing operators

The **narrowing operators** (also called inverse operators) are analogous to the basic interval operators used for evaluating binary and unary expressions (see Section 2.2). They allow us to *optimally* compute the projections associated to primitive constraints (e.g., $w = x_1 + x_2$, $w = x_1 \times x_2$, $w = sin(x)$, $w = log(x)$). Like arithmetic operators we distinguish two types: the unary operators (related to binary constraints $w = f(x)$, where $f \in \{cos, sin, log, ...\}$), and the binary operators (related to ternary constraints $w = x_1 \circ x_2$ where $\circ \in \{+, -, \times, /\}$).

Definition 12 (Unary narrowing operator) *Let* $c : w = f(x)$ *be a constraint.* f *is a primitive function (i.e.,* $f \in \{cos, sin, log, ...\}$*)* $[x]$ *and* $[w]$ *are the related domains of variables.* N_x^f *is the narrowing operator of* f *over* x *if:*

$$N_x^f([w], [x]) = \texttt{Hull}(\Pi_x^c([w], [x]))$$

The narrowing operator of a unary function f can be obtained from the generated expression by isolating the variable x from the constraint c. For example, the narrowing operator for the square operator ($c : w = x^2$) is $N_x^{x^2}([w], [x]) = [x] \cap \texttt{Hull}(-\sqrt{[w]}, \sqrt{[w]})$. When the `Hull` operator is used before the intersection, it is possible to obtain an overestimation of the projection. Thus, it is advisable to perform the computation using the union operator and to apply the hull only after the intersection. In the case of the square operator, the narrowing operator is modified to:

$$N_x^{x^2}([w], [x]) = \texttt{Hull}([x] \cap -\sqrt{[w]}, [x] \cap \sqrt{[w]})$$

Consider the intervals $[x] = [4, 10]$ and $[w] = [25, 36]$. The narrowing using the hull operator before the intersection computes a non-optimal projection $N_x^{x^2}([w], [x]) = [4, 10] \cap \texttt{Hull}([-6, -5], [5, 6]) = [4, 6]$. The optimal projection is obtained using the union before the hull operator:

$$N_x^{x^2}([w], [x]) = \texttt{Hull}(([4, 10] \cap [-6, -5]) \cup ([4, 10] \cap [5, 6])) = [5, 6].$$

Definition 13 (Binary narrowing operator) *Let $c(w, x_1, x_2) : w = f(x_1, x_2)$ be a constraint. $[x_1]$, $[x_2]$ and $[w]$ are the domains of the variables. The narrowing operators of f over x_1 and x_2 are:*

$$N^f_{x_1}([w], [x_1], [x_2]) = \texttt{Hull}(\Pi^c_{x_1}([w], [x_1], [x_2]))$$
$$N^f_{x_2}([w], [x_1], [x_2]) = \texttt{Hull}(\Pi^c_{x_2}([w], [x_1], [x_2]))$$

The binary narrowing operators are defined for the four basic operators (i.e., $+$, $-$, $\times$, $/$). They can be obtained by finding the roots of x_1 (and x_2) in function of the other variables of the related primitive constraint. Then, the roots are evaluated using the natural evaluation and intersected with the current domain of variables. For example, the narrowing operators related to the product ($\times$) are deduced from the constraint $w = x_1 \times x_2$[1]:

$$N^{x_1 \times x_2}_{x_1}([w], [x_1], [x_2]) = [x_1] \cap ([w]/[x_2])$$
$$N^{x_1 \times x_2}_{x_2}([w], [x_1], [x_2]) = [x_2] \cap ([w]/[x_1])$$

In the same way as for unary narrowing operators, the evaluation of the expression $[x_1]/[w]$ should consider the exact image set and not only the hull.

Example 24 *Consider the constraint $c(x, y, z) : (x - y)^2 = z$, with domains $[x] = [8, 10]$, $[y] = [0, 4]$ and $[z] = [5, 6]$. The algorithm* `HC3-Revise` *performs a decomposition of c into the set of primitive constraints:*

$$\begin{array}{ll} c'_1(x, y, w_1): & w_1 = x - y \\ c'_2(w_1, w_2): & w_2 = w_1^2 \\ c'_3(z, w_2): & z = w_2 \end{array}$$

The initial domains of the auxiliary variables w_1 and w_2 are $[-\infty, +\infty]$. The domains of $[w_1]$ and $[w_2]$ are intersected by the natural evaluation of the right side of the first two constraints, i.e., $[w_1] = [-\infty, +\infty] \cap [8, 10] - [0, 4] = [4, 10]$ and $[w_2] = [-\infty, +\infty] \cap [4, 10]^2 = [16, 100]$. In c'_3, $[w_2]$ is intersected with the domain of $[z]$, then $[w_2] = [16, 100] \cap [25, 36] = [25, 36]$. The reduction of $[w_2]$ is propagated to c'_2, then $[w_1]$ is reduced by the corresponding narrowing operator:

$$[w_1] = N^{w1^2}_{w_1}([w2], [w1]) = [4, 10] \cap ([-6, -5] \cup [5, 6]) = [5, 6]$$

Finally the reduction of $[w_1]$ is propagated to the constraint c'_1:

$$[x] = N^{x-y}_x([w_1], [x], [y]) = [8, 10] \cap ([5, 6] + [0, 4]) = [8, 10]$$
$$[y] = N^{x-y}_y([w_1], [x], [y]) = [0, 4] \cap ([8, 10] - [5, 6]) = [2, 4]$$

`HC3-Revise` enforces the hull-consistency in the set of primitive equations (if the constraint has not the trivial expressions xx, $x - x$ nor x/x). Due to the locality scope of local consistencies (reduction of one constraint at a time), enforcing the hull-consistency in the set of primitive constraints results in a larger box than (or equal to) enforcing the hull-consistency in the original constraint. In particular, if the original (non primitive) constraint c has no multiple occurrence of variables (and the function is continuous and bijective in the domain $[B]$), `HC3-Revise` enforces the hull-consistency of c.

[1]The implementation should also consider some pathological cases that occur when $0 \in [x_1]$, $0 \in [x_2]$ and/or $0 \in [w]$.

3.2.3.4 The algorithm HC4-Revise

As we have mentioned, like the HC3-Revise algorithm, HC4-Revise also enforces the hull-consistency of primitive constraints. However, HC4-Revise performs an *automatic projection* of the constraint c over all its variables avoiding the decomposition of c into primitive constraints. The algorithm, motivated by automatic differentiation (see Section 2.2.4.2), uses a tree representation of the constraint, where the leaves are constants or variables, and internal nodes correspond to the basic operators. An interval is associated to every node.

HC4-Revise works in two phases. The **evaluation** phase is recursively performed from the leaves to the root. This phase computes, using the interval operators of interval arithmetic, the natural evaluation of the subexpressions represented by the tree nodes. Consider for example the node $w_2 = w_1^2$. The related interval evaluates $[w_1]^2 = [4, 10]^2 = [16, 100]$. The **narrowing** phase traverses the tree top-down from root to leaves and applies in every node the related narrowing operator (see Figure 3.1-right). The narrowing operator reduces the intervals of the nodes eliminating inconsistent values w.r.t. the corresponding unary or binary basic operator. In Figure 3.1, the intervals in bold have been narrowed. If an empty interval is obtained during the narrowing phase, this means that the initial domains do not satisfy the constraint, then an empty box is returned. Consider for example, the node w_1 corresponding to the sub-expression $w_1 = x - y$ in Figure 3.1-right. The projection over y is performed by the narrowing operator

$$N_y^{x-y}([w_1], [x], [y]) = [0, 4] \cap ([8, 10] - [5, 6]) = [2, 4]$$

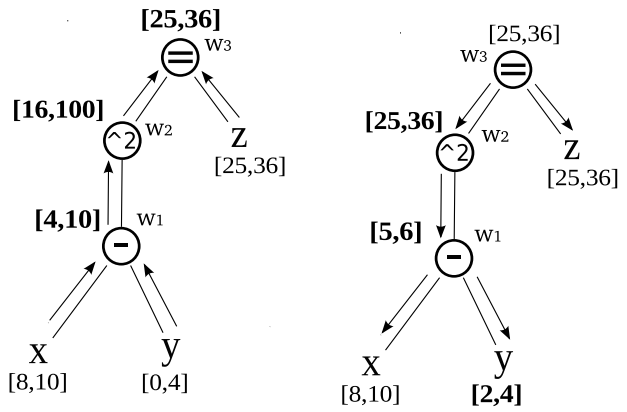

Figure 3.1: Evaluation and projection phases of the HC4-Revise procedure in the constraint $(x - y)^2 = z$ with domains $[x] = [8, 10]$, $[y] = [0, 4]$ and $[z] = [25, 36]$.

HC3-Revise and HC4-Revise compute the same narrowed box. However, HC4-Revise is more efficient [Benhamou et al., 1999].

Let e be the number of nodes in the expression tree, i.e., e is the number of basic (binary or unary) operators in the function. The time complexity of the HC4-Revise procedure is $O(e)$.

HC4-Revise performs the natural evaluation of the projection functions related to each occurrence automatically (i.e., it avoids to generate symbolically each projection). It is trivial to deduce then, that if a constraint f is continuous and bijective in the domain and each variable appears only once in f then HC4-Revise computes the hull-consistency of the constraint $f(X) = 0$ (as the variables appear only once in each projection function, the natural evaluation is optimal, see Section 2.4.1).

A combinatorial variant of the algorithm, called TAC-Revise [Chabert, 2007; Chabert et al., 2005] enforces the hull-consistency allowing discontinuities in the function and in the corresponding projection functions.

In `TAC-Revise` all the operators compute the exact image of the related intervals instead of performing the hull (the images are maintained as a union of several intervals adding the consequent combinatorial aspect to the computations). At the end of the narrowing phase, for each variable, the obtained union of intervals is hulled and the resulting box is returned.

In practice, the additional work performed by `TAC-Revise` is often useless and `HC4-Revise` computes the same box as `TAC-Revise`. After a few bisections, i.e., very high in the search tree, the functions generally become continuous in the current search space.

On the other hand, when the constraint has multiple occurrences of variables, it is necessary to use other procedures for improving the contraction.

3.2.3.5 The `Box-Revise` algorithm

The `Box` algorithm [Benhamou et al., 1999, 1994; Van Hentenryck et al., 1997] increases the contraction power of `HC4-Revise` in a constraint c by calling a stronger contraction procedure called `BoxNarrow` (or `Box-Revise`) on each variable appearing several times in c. It was the main constraint propagation algorithm used in `Numerica` [Van Hentenryck et al., 1997].

In particular, if only one variable x appears several times in a constraint c (and the constraint is continuous in the studied box), then the `BoxNarrow` procedure *generally*[1] computes the optimal projection of c over x (with a given precision ϵ).

Definition 14 (Box-consistency) *Consider the equation $c : f(x_1, ..., x_k) = 0$. The pair (c, x_i) is box-consistent in the box $[B] = [x_1] \times ... \times [x_k]$ if:*

$$0 \in [f]_n([x_1], ..., [x_{i-i}], [\underline{x_i}, \underline{x_i}^+], [x_{i+1}], ...[x_k]);$$
$$0 \in [f]_n([x_1], ..., [x_{i-i}], [\overline{x_i}^-, \overline{x_i}], [x_{i+1}], ...[x_k]).$$

a^+ (resp. a^-) is the smallest (resp. largest) floating point number greater than (resp. smaller than) a. $[f]_n$ is the natural extension of f.

The `BoxNarrow` procedure enforces the box-consistency for each variable appearing several times in the function. Consider the constraint $c : f(x_1, ..., x_k) = 0$. For enforcing the box-consistency of the pair (f_i, x_i) `BoxNarrow` works with the interval function:

$$[f_i](x_i) = [f]_n([x_1], ..., [x_{i-i}], x_i, [x_{i+1}], ..., [x_k])$$

Observe that all the variables, excepting x_i, have been replaced by their domains, i.e., they are constants. In Figure 3.2 we can see an example of the interval function $[f_i]$. `BoxNarrow` tries to calculate the smallest interval containing all the zeros of this function (black line). After a dichotomic process, the procedure is able to obtain a thin/sharp approximation of the leftmost and rightmost *quasi-zeros* (specifically, `BoxNarrow` computes two intervals $[l]$ and $[r]$ of diameter one u.l.p. such that $0 \in [f_i]([l])$ and $0 \in [f_i]([r])$). At the end, the algorithm contracts the domain of $[x_i]$ to $[\underline{l}, \overline{r}]$. Note that due to floating point imprecisions, interval-based methods cannot predict that the atomic interval $[r]$ in the figure ($[r] = [\overline{r}^-, \overline{r}]$) does not contain a zero of the function, buat a quasi-zero.

`BoxNarrow` executes two procedures for obtaining the interval $[l, r]$: one for finding the leftmost quasi-zero of $[f_i](x)$ (`LeftNarrow`) and the other for finding the rightmost quasi-zero (`RightNarrow`). Algorithm

[1] Excepting cases related to quasi-zeros of the function (see Figure 3.2).

3 describes an implementation of the `LeftNarrow` procedure (based on the `BC3-Revise` version of the algorithm described in [Benhamou et al., 1999; Van Hentenryck et al., 1997]).

Algorithm 3 `LeftNarrow`(**in:** $[x_i]$, $[f_i]$, f)

$[l] \leftarrow [x_i]$
if $0 \notin [f_i]([l])$ **then return** $\emptyset$
$[l] \leftarrow$ `univariateNewton`$([f_i], \frac{\partial f}{\partial x_i}, [l])$
if $\overline{l} \leq \underline{l}^+$ **then return** $[l]$ /* $[l]$ is atomic */
if $[l] = \emptyset$ **then return** $\emptyset$
$([l_1], [l_2]) \leftarrow$ `Bisect`$([l])$
$[l] \leftarrow$ `LeftNarrow`$([l_1], [f_i], f)$
if $[l] = \emptyset$ **then return** `LeftNarrow`$([l_2], [f_i], f)$
return $[l]$

The interval $[l]$, used for finding the leftmost quasi-zero, is initialized with the domain of $[x_i]$. The search is recursively performed in a "dichotomic" way aided with Newton steps for performing additional contraction to $[l]$, thus accelerating the process. If all the values of the current interval are inconsistent, then the algorithm returns an empty interval. If the diameter of the current interval $[l]$ is less than or equal to 1 u.l.p. and $0 \in [f_i]([l])$, then the procedure returns $[l]$.

The `univariateNewton` procedure computes a fixpoint of the univariate Interval Newton algorithm (described in Section 3.2.1). In each iteration, the procedure performs the evaluations of $[f_i](\texttt{Mid}[\texttt{l}])$ and $[\frac{\partial f}{\partial x_i}]([l])$. The contraction of $[l]$ performed in an iteration of Newton is given by:

$$[l] \leftarrow [l] \cap \left(\texttt{Mid}([l]) - \frac{[f_i](\texttt{Mid}([l]))}{[\frac{\partial f}{\partial x_i}]([l])} \right) \tag{3.9}$$

Figure 3.2 shows the steps followed by the dichotomic process. The top (resp. the bottom) side of the figure details the work performed by `LeftNarrow` (resp. `RightNarrow`). The last step (17) achieved by `BoxNarrow` replaces the interval $[x]$ by the new interval $[\underline{l}, \overline{r}]$.

In practice, the `Box` algorithm obtains some good results, especially when the constraints have only one variable appearing several times. In general, when several variables appear several times, the additional work of `BoxNarrow` w.r.t. `HC4-Revise` is not rewarded enough by the additional contraction.

One of the main contributions reported in this thesis is the proposition of a new contractor, called `Mohc-Revise`, aiming at the general case where several variables appear several times. This algorithm uses a more effective variant of `BoxNarrow` called only when the function is monotonic w.r.t. the variable in the current box (described in Chapter 4).

Remark that the `LeftNarrow` and `RightNarrow` procedures are not really dichotomic: the complexity in the worst case is time $O(d)$ with d the number of floating points between $\underline{x}$ and $\overline{x}$ (if a precision ϵ, expressed as a ratio of the initial diameter of $[x_i]$, is given then $d = 1/epsilon$).

3.2.3.6 Other filtering techniques for enforcing hull-consistency

In this section we describe the algorithm `Octum` proposed in [Chabert and Jaulin, 2009b]. This algorithm presents some similarities with a procedure of our algorithm `Mohc` presented in Chapter 4. `Octum` and `Mohc` have been initiated independently in the first semester of 2009.

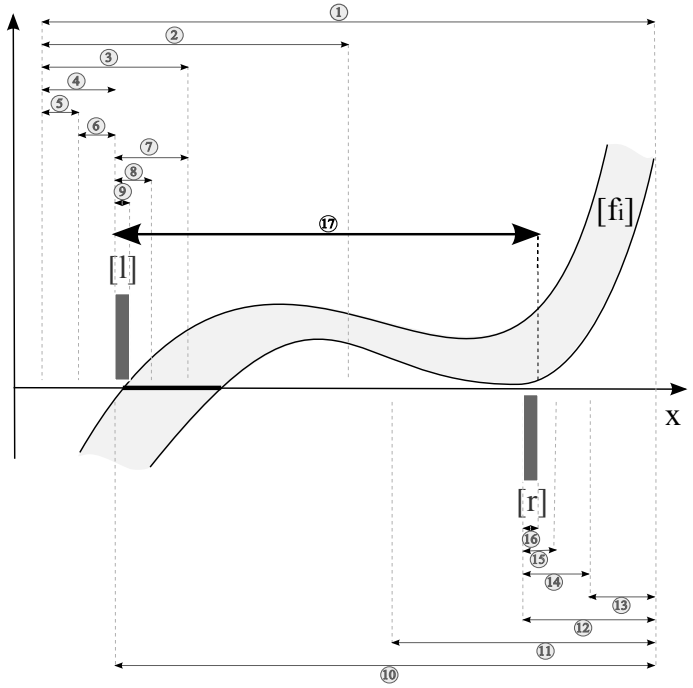

Figure 3.2: The `BC3-Revise` procedure. The algorithm returns the interval computed in the step 17. (To simplify the figure, the additional contraction performed by Newton is not considered.)

`Octum` is a polynomial-time algorithm that enforces hull-consistency in a *monotonic* function f (i.e., that optimally projects over all the variables of f). We say that a function f is monotonic, when the partial derivatives of f w.r.t. each variable is positive (resp. negative) in every element of the domain (for more details and basic concepts related to the monotonicity, see Section 2.4.2).

`Octum` allows us to extend the cases of optimal projection to monotonic functions with several occurrences of variables. The algorithm is based on the extension by monotonicity (this extension computes an optimal image of monotonic functions, see Section 2.4.2) and on the *idempotence of the projection* (Section 3.2.3.2), i.e., it is enough to project (optimally) once over each variable of f to obtain the hull-consistency.

The `Octum` algorithm uses a `BoxNarrow` procedure for contracting the domain of the variables ($[B] = [x_1] \times ... \times [x_k]$) by the constraint $f(X) = 0$. Consider the procedure `LeftNarrow` of `Octum` (that we call `LeftNarrow-Octum`) applied to the interval $[x_i]$. Recall that the goal of the `LeftNarrow` procedure is to find the leftmost zero in $[x_i]$, i.e., the minimum point $l \in [x_i]$ satisfying the relation $0 \in [f]([x_1], ..., [x_{i-1}], [\underline{l}, \underline{l}^+], [x_{i+1}], ..., [x_k])$ (where $[f]$ is generally the natural extension). `Octum` uses the evaluation by monotonicity instead of the natural extension. One important advantage of using the evaluation by monotonicity is that it is optimal for monotonic functions, i.e., $0 \in [f]_m([x_1], ..., [x_{i-1}], [\underline{l}, \underline{l}^+], [x_{i+1}], ..., [x_k])$ implies that the interval $[l, \overline{x_i}]$ is *hull-consistent* on the left bound of x_i.

Instead of working with an interval function for contracting the left bound of the interval $[x_i]$ (symmetrically for the right bound), `Octum` uses the *punctual* function:

$$\overline{f_i}(x_i) = f(x_1^+, ..., x_{i-1}^+, x_i, x_{i+1}^+, ..., x_k^+),$$

if x_i is increasing and the symmetric function:

$$\underline{f_i}(x_i) = f(x_1^-, ..., x_{i-1}^-, x_i, x_{i+1}^-, ..., x_k^-),$$

if x_i is decreasing, where $[B^+] = (x_1^+, ..., x_k^+)$ and $[B^-] = (x_1^-, ..., x_k^-)$ are the boxes used by the evaluation by monotonicity to compute resp. the left and right bound of $[f]_m([B])$ (see Definition 3, page 25). As

the function is monotonic, all the intervals in $[B^-]$ (and $[B^+]$) are degenerate. $\overline{f_i}(x_i)$ corresponds to the maximum of the image of $[B]$ under f in function of the variable x_i (resp. $\underline{f_i}(x_i)$ corresponds to the minimum of the image). See an example in Figure 3.3. The function $[f_i](x) = [\underline{f_i}(x), \overline{f_i}(x)]$ (i.e., the thick gray curve on the figure; the upper black border represents the function $\overline{f_i}(x_i)$ whereas the lower border represents $\underline{f_i}(x_i)$) corresponds to the *optimal* image of $[B]$ under f in function of x_i.

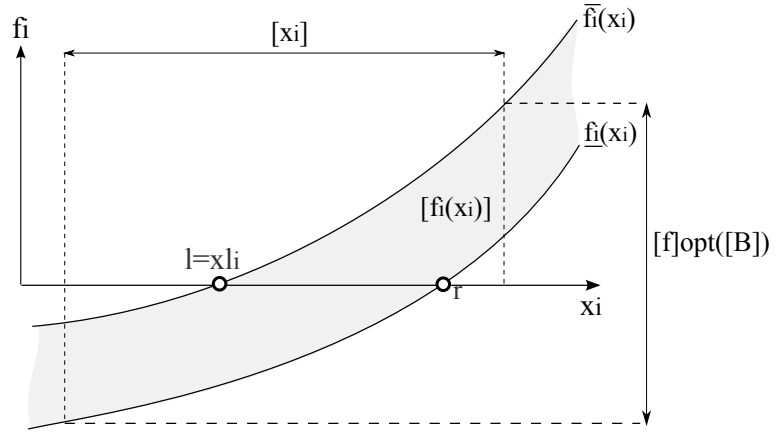

Figure 3.3: The functions used by the `LeftNarrow-Octum` procedure.

For finding l, the univariate interval Newton method searches the zero of $\overline{f_i}(x)$ in the interval $[x_i]$. As f_i is a real function and is monotonic w.r.t. x in the interval $[x_i]$, the method converges to l in $O(log(1/\epsilon))$ (where ϵ is the desired precision expressed as a ratio of the initial diameter of $[x_i]$) if we use the midpoint in the Newton iteration (as in (3.9) page 51).

Furthermore, using the following proposition based on monotonicity properties, `Octum` avoids projecting over all the variables of one constraint.

Proposition 8 *Let f be a continuous and monotonic function in the box $[B] = [x_1] \times .... \times [x_k]$ such that the image of $[B]$ under f contains zero, i.e., $0 \in \mathcal{I}f([B])$. Assume that f is increasing w.r.t. x_i. If xl_i is the leftmost zero in the interval $[x_i]$, i.e.,*

$$xl_i = \inf\{v, v \in [x_i], 0 \in \mathcal{I}f([x_1], ..., [x_{i-1}], v, [x_{i+1}], ..., [x_k])\},$$

then one of two options holds:

1. $xl_i = \underline{x_i}$
2. $f(x_1^+, ..., x_{i-1}^+, xl_i, x_{i+1}^+, ..., x_k^+) = 0.$

Where for all $j \neq i$, $x_j^+ = \overline{x_j}$ if x_j is increasing and $x_j^+ = \underline{x_j}$ if x_j is decreasing.

The proof is described in [Chabert and Jaulin, 2009b].

Proposition 8 indicates that if the optimal projection of a monotonic function f over an increasing variable x_i reduces the domain $[x_i]$, then an optimal projection cannot reduce one bound of any other variable x_j (the left bound if x_j is decreasing and the right bound if it is increasing). The proposition is used for avoiding some calls to `LeftNarrow-Octum` and `RightNarrow-Octum` procedures.

In Chapter 4 we present our algorithm `Mohc` containing a similar procedure (i.e., `MonotonicBoxNarrow`) for narrowing monotonic variables.

3.2.4 Strong consistency algorithms

The locality problem, described in Section 3.6, is intrinsic to local consistencies and can be reduced by enforcing stronger consistencies. Contractors enforcing stronger consistencies take into account more than one constraint at a time to perform filtering.

3.2.4.1 The 3B algorithm

Definition 15 (3B-consistency) *Consider a NCSP $P = (C, X, [B])$ with a set of constraints $C = \{c_1, ..., c_k\}$ and a set of variables $X = \{x_1, ..., x_k\}$. A variable x_i is 3B-consistent in $[B] = [x_1] \times ... \times [x_k]$ if:*

$$\text{the system is hull-consistent in the box } [B] = [x_1] \times ... \times [x_{i-1}] \times [\underline{x_i}, \underline{x_i}^+] \times [x_{i+1}] \times ... \times [x_k]$$
$$\text{the system is hull-consistent in the box } [B] = [x_1] \times ... \times [x_{i-1}] \times [\overline{x_i}^-, \overline{x_i}] \times [x_{i+1}] \times ... \times [x_k]$$

a^+ (resp. a^-) is the smallest (resp. largest) floating point number greater than (resp. smaller than) a. A system is 3B-consistent if all the variables are 3B-consistent.

The contraction algorithm 3B, proposed by Olivier Lhomme [Lhomme, 1993] during his Phd thesis with Michel Rueher, enforces a weak form of 3B-consistency. The procedure basically consists in splitting the domain of a variable x_i in several slices. In each slice an algorithm enforcing a weak form of hull-consistency is applied (e.g., HC4). Each inconsistent slice is removed from the domain of x.

The basic 3B algorithm is described in Algorithm 4.

Algorithm 4 3B(**in-out:** $[B]$; **in:** C, X, SubContractor, ϵ_{outer}, ϵ_{inner})

```
[B_0] ← [B]
repeat
  for all i = 1..k do
    VarShaving([B], C, X, i, Subcontractor, ε_inner)
  end for
until StopCriterion(ε_outer, [B], [B_0])
```

The input of the 3B procedure includes the system (i.e., the set of constraints C and the set of variables $X = \{x_1, ..., x_k\}$) and the current box $[B]$. It also requires 3 user parameters: Subcontractor which is a contraction procedure (e.g., HC4, Box called in a given slice); the parameters ϵ_{outer} and ϵ_{inner} which are interval diameters allowing stopping the algorithm when a certain precision is reached. We design by 3B(HC4) (resp. 3B(Box)) the 3B algorithm parameterized with the HC4 (resp. Box) subcontractor. In Numerica [Van Hentenryck et al., 1997] the algorithm 3B(Box) is called BoundConsistency.

The for all loop processes all the variables iteratively. The StopCriterion procedure tests whether one interval in $[B]$ has been reduced more than ϵ_{outer}; if this is the case the process is repeated.

The heart of the algorithm is the VarShaving procedure. This procedure (Algorithm 5) performs the contraction of the interval bounds of a variable x_i using the subcontractor. VarShaving attempts to narrow $[x_i]$ on the left side by dividing the interval into r slices $[s_j]$ of size ϵ_{inner}. Then, the slices are processed iteratively from left to right. For each slice $[s_j]$ a new box $[B']$ is generated by replacing the interval $[x_i]$ of $[B]$ by $[s_j]$. The subcontractor is applied to the system using $[B']$ as the initial box. If

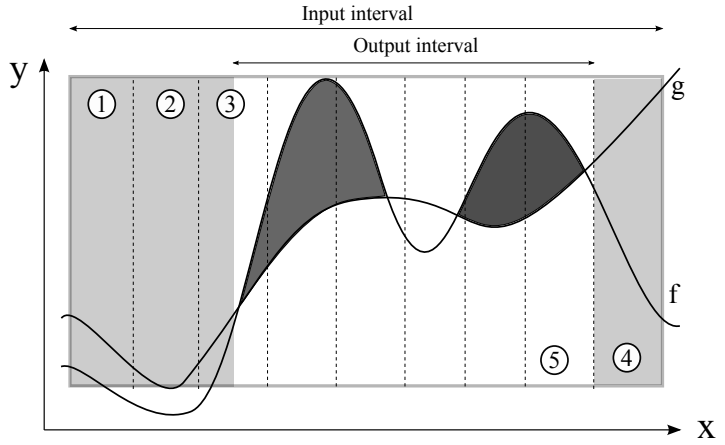

Figure 3.4: The `VarShaving` procedure applied to a variable x.

the subcontractor returns an empty box, then the slice is removed from $[x_i]$ (i.e., the left bound of the interval is updated: $\underline{x_i} \leftarrow \overline{s_j}$). Otherwise, the slice has only been contracted (not removed). The left bound of $[x_i]$ is updated with the left bound of the contracted slice (i.e., $\underline{x_i} \leftarrow \underline{[B']_i}$) and the loop stops. The contraction on the right bound is performed in a similar way.

Figure 3.4 shows an example of the algorithm applied to a variable x. The solution is represented by the dark gray area above the curve g and below the curve f. The system is hull-consistent in the initial domains (each constraint is satisfied at the bounds of the box). The domain of the variable x is split into slices by the `VarShaving` procedure. The first two processed slices from left to right are discarded by the subcontractor and are removed from the interval $[x]$. The third slice is only reduced, implying a smaller reduction in $[x]$. The process is repeated symmetrically from right to left.

Remarks

The `3B` algorithm is not an incremental procedure, i.e., the `repeat` loop performs the reduction over all the variables, because the reductions obtained by the subcontractor are caused by the entire system (that is why it is a strong consistency algorithm). `3B(propag)` brings a better filtering than `propag`, i.e., the box obtained by the former is contained in (or equal to) the box obtained by the latter.

The generalization of the 3B-consistency (see Definition 15) is recursive, i.e., the `kB algorithm` is equivalent to Algorithm 4 using as subcontractor the `(k-1)B algorithm`. For example the `4B` uses as subcontractor the `3B` algorithm.

The `3B` algorithm is a polynomial time algorithm. `3B` performs choice points splitting the intervals, similarly to bisections. However, the obtained reductions are hulled in only one interval (i.e., `VarShaving` returns a single box), avoiding the combinatorial explosion of the search tree.

The principle of `VarShaving` is similar to the procedure `BoxNarrow`. In fact, similarly to `BoxNarrow`, a more classical implementation of `VarShaving` builds the slices in a dichotomic way. The main difference is that `BoxNarrow` can only remove slices by using one constraint. `VarShaving` uses a constraint propagation on all the system for trying to remove a slice related to a variable x. The cost for refuting a given slice is also low if an incremental propagation is run with only the constraints involving x. This is an interesting property that explains the generally better performance obtained by versions of the `3B` algorithm.

Algorithm 5 `VarShaving`(**in-out:** $[B] = \{[x_1], ..., [x_k]\}$; **in:**$C$, X, i `SubContractor`,ϵ_{inner})

$([s_1], ..., [s_r]) \leftarrow$ `Split`$([x_i], \epsilon_{inner})$
/* The slices are treated from left to right ($s_1 \rightarrow s_r$) */
for all $j = 1..r$ **do**
  $[B'] \leftarrow \{[x_1], ..., [x_{i-1}], [s_j], [x_{i+1}], ..., [x_k]\}$
  $[B'] \leftarrow$ `SubContractor`$([B'], C, X)$
  if $[B'] = \emptyset$ **then**
    $\underline{x_i} \leftarrow \overline{s_j}$ /* the slice is removed */
    if $i = k$ **then return** $\emptyset$ **end if** /* all the slices have been removed */
  else
    $\underline{x_i} \leftarrow \underline{[B']_i}$ /* the slice has only been reduced */
    break
  end if
end for

/* The slices are treated from right to left ($s_r \rightarrow s_{j+1}$) */
for all $j' = r..j + 1$ **do**
  $[B'] \leftarrow \{[x_1], ..., [x_{i-1}], [s_{j'}], [x_{i+1}], ..., [x_k]\}$
  $[B'] \leftarrow$ `SubContractor`$([B'], C, X)$
  if $[B'] = \emptyset$ **then**
    $\overline{x_i} \leftarrow \underline{s_{j'}}$ /* the slice is removed */
  else
    $\overline{x_i} \leftarrow \overline{[B']_i}$ /* the slice has only been reduced */
    break
  end if
end for

3.2.4.2 The `3BCID` algorithm

The algorithm `3BCID` is an improved version of `3B`. It has been proposed in [Trombettoni and Chabert, 2007] and it is based on the *constructive disjunction* principle used to treat an extension of the CSP model to disjunctive constraints. This idea can also be applied to a finite domain of a classical CSP, where each domain can be represented by a disjunction of unary constraints.

Constructive Interval Disjunction (CID) implements this domain constraint disjunction over intervals. The algorithm is close to the `3B` algorithm excepting that the `VarShaving` procedure is replaced by a `VarCID` procedure. The `VarCID` procedure, applied to an interval $[x]$ (see Figure 3.5-a), splits the initial box into s subboxes (in the figure, $s = 9$). Each subbox is contracted using a filtering algorithm (e.g., `HC4`). The final contraction brought by `VarCID` is obtained by performing the *hull* of the reduced subboxes. Observe that the contraction on $[x]$ is equivalent to the one performed by `VarShaving`. However, `VarCID` adds an additional contraction on $[y]$.

The `3BCID` algorithm uses a procedure called `Var3BCID` that combines the features of the two procedures: `VarShaving` and `VarCID`. It is parameterized with a precision ϵ_{inner} and with a number of slices s_{cid}. The principle is described below and is illustrated in Figure 3.5-b:

1. A shaving procedure is performed over the interval $[x]$ using the parameter ϵ_{inner} (in the figure: steps 1-3 on the left bound and 4-5 on the right bound of the interval). The only difference with `VarShaving` is that the last contracted subboxes ($[B_l]$ and $[B_r]$) are saved.

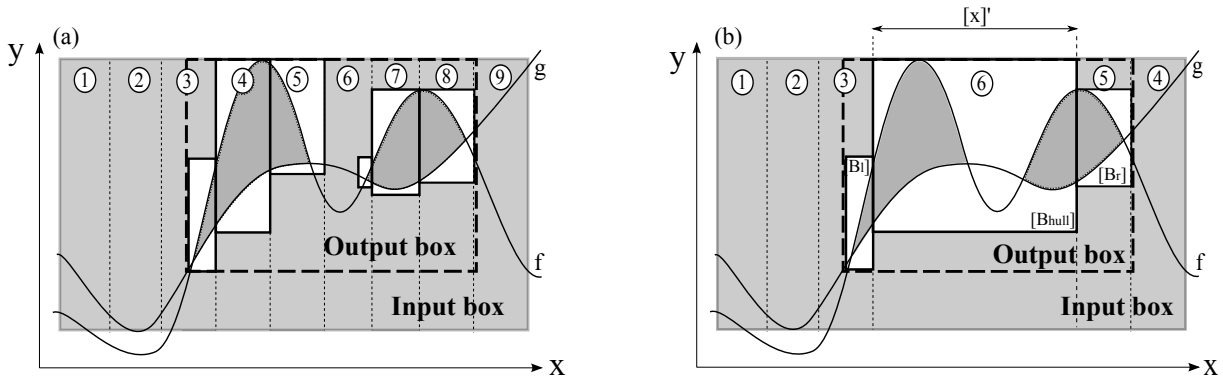

Figure 3.5: (a) The `VarCID` procedure applied to a variable x. (b) The `Var3BCID` procedure applied to a variable x.

2. The remaining interval $[x]'$ is treated by the `VarCID` procedure, i.e., the interval is split in at most s_{cid} slices that are contracted using the subcontractor. The procedure returns the hull of the contracted boxes $[B_{hull}]$. In the figure $s_{cid} = 1$, then the subcontractor has been applied to all the remaining box.

3. The algorithm returns:
$$\texttt{Hull}([B_l], [B_{hull}], [B_r])$$

In the example of the figure, the algorithm `3BCID` returns the same box that `CID` while performing less steps.

Note that `3B` is equivalent to `3BCID` using $s_{cid} = 0$. The authors advise to fix the parameter s_{cid} to 1 [Trombettoni and Chabert, 2007]. In practice, the contraction performed by `3BCID` is comparable to the one performed by `3B`, however it generally reaches the (quasi) fixpoint faster due to the contraction in several dimensions performed by `VarCID`. In other terms, the total number of calls to `Var3BCID` is generally even inferior to the number of calls to `VarShaving` for obtaining a same contraction.

In this thesis, `3BCID` is the main filtering algorithm used to compare the results of our implementations. `3BCID(Mohc)` seems also to be a very promising contractor based on our new `Mohc` algorithm.

3.2.5 Global Hull Consistency and the locality problem

Definition 16 (Global Hull Consistency) *Consider the NCSP $P = (C, X, [B])$. P is global hull-consistent if $[B]$ is the smallest box containing all the solutions of P.*

The **locality problem** generally prevents from obtaining a global hull-consistent box by local filtering algorithms. The problem is due to the reduced scope of constraint propagation techniques. For instance, Figure 3.6-left illustrates a system made of two constraints c_1 and c_2. The system is hull-consistent because each constraint is hull-consistent and cannot be reduced independently. Stronger consistency techniques (in particular kB-consistencies) can partially solve this problem (Figure 3.6-right). However in practice, when the systems have several variables and constraints, enforcing kB-consistency with $k > 3$ is computationally expensive.

Interval Newton methods deal with this problem too. These methods treat the whole system as one global constraint and perform the global hull-consistency using a relaxed linear system. However, their

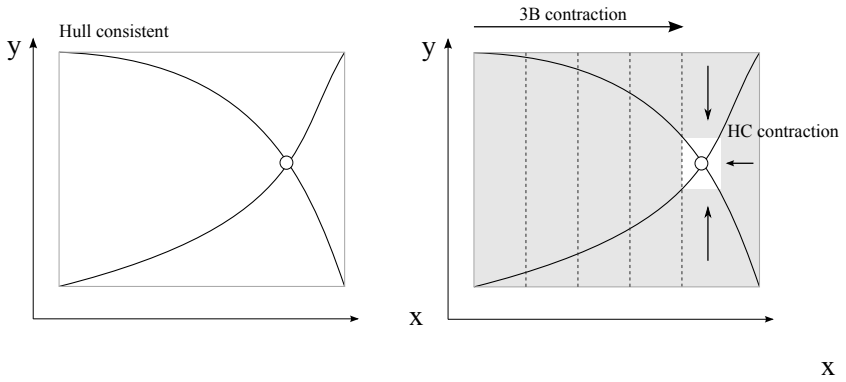

Figure 3.6: Example of locality problem. On the right a hull-consistent system affected by the locality problem. On the left, `3B` and hull-consistency algorithms are performed and reduce the locality problem.

effectiveness is limited by the characteristics of the system in the current box $[B]$: Interval Newton can only converge in square well-constrained systems with only one solution in $[B]$, when $[B]$ is sufficiently small.

3.3 Splitting Algorithms

After having performed the contraction of the current box, the *splitting* algorithms are in charge of dividing the box into two or more subboxes, thus continuing the combinatorial solving process.

Definition 17 (Splitting) *Consider the box* $[B]$. *A* **splitting operator** *divides* $[B]$ *into a set of* n *subboxes* $\{[B_1], ..., [B_n]\}$ *such that:*

$$[B] = [B_1] \cup .... \cup [B_n]$$

In particular, a **bisection** operator is a splitting operator that divides the box into only two subboxes.

Example 25 *Consider the box* $([1,3],[0,6])$. *If we divide the box in the middle of each interval we obtain 4 subboxes*

$$[B_1] = ([1,2],[0,3]); \quad [B_2] = ([2,3],[0,3]); \quad [B_3] = ([1,2],[3,6]); \quad [B_4] = ([2,3],[3,6])$$

The obtained boxes satisfy the relation: $[B] = [B_1] \cup [B_2] \cup [B_3] \cup [B_4]$

Consider a problem $P = (C, X, [B])$. In general, each subproblem $P_i = (C, X, [B_i])$ is easier to solve because the problem has no solutions or because some filtering algorithm (e.g., `interval Newton`, `3BCID`) is more effective (recall that smaller domains of variables imply a reduction of the dependency and locality problems).

A splitting/bisection consists of two steps: the selection of the variable(s) to be split and the selection of the value(s) in which the box will be split. In the next section we describe the main heuristics used for performing these choices.

3.3.1 Variable selection

An splitting is preceded by the selection of the variable(s) to be split. The most common methods select only one variable. In the following, we list the most commonly used strategies for determining the variable to be split.

Round-Robin: It is one of the simplest and most used methods. It does not require any information on the system. If the current k-dimensional box $[B]$ was obtained by bisecting the interval $[x_i]$ then the round-robin method will select the variable x_j, with j= $(i+1)$ *mod* k. The objective is to not forget any variable. One weakness of this strategy is that a *bad* initial order of variables can lead to disastrous performance.

Largest-First: The largest domain is bisected first. It is based on the assumption that intervals with large diameters have a greater influence on the function evaluation. In the same way as the Round-Robin technique, the largest-first method does not forget any variable.

Smear-based [Kearfott and Novoa III, 1990]: These methods use the information of the system for obtaining the most important variable w.r.t. a criterion. The criterion reflects the *impact* of a variable x over all functions that involve x and can be deduced from the Taylor extension (see Section 2.4.3):

$$[f]_t([B]) = f(\texttt{Mid}([B]) + \sum_{i=1}^{k} \left[\frac{\partial f}{\partial x_i}\right]([B])([x_i] - \texttt{Mid}([x_i]))$$

Observe that the diameter of $[f]_t([B])$ can be computed in function of the diameter of the interval partial derivatives. Knowing that:

- $\texttt{Diam}(f(\texttt{Mid}([B]))) = 0$,
- $\texttt{Diam}\left(\sum_{i=1}^{k}[w_i]\right) = \sum_{i=1}^{k}\texttt{Diam}([w_i])$,
- $[x_i] - \texttt{Mid}([x_i]) = [-\frac{\texttt{Diam}([x_i])}{2}, \frac{\texttt{Diam}([x_i])}{2}]$ and
- $\texttt{Diam}([\frac{\partial f}{\partial x_i}]([B]) \times [-a/2, a/2]) = \texttt{Diam}([-|[\frac{\partial f}{\partial x_i}]([B])| \times a/2, |[\frac{\partial f}{\partial x_i}]([B])| \times a/2] = |[\frac{\partial f}{\partial x_i}]([B])| \times a$,

we deduce:

$$\texttt{Diam}([f]_t([B])) = \sum_{i=1}^{k}\texttt{Diam}\left(\left[\frac{\partial f}{\partial x_i}\right]([B]))([x_i] - \texttt{Mid}([x_i]))\right) = \sum_{i=1}^{k}\left|\left[\frac{\partial f}{\partial x_i}\right]([B])\right| \times \texttt{Diam}([x_i])$$

Thus, the impact of the variable x_i on the diameter of $[f]_t([B])$ is $s(x_i, f, [B]) = |[\frac{\partial f}{\partial x_i}]([B])| \times \texttt{Diam}([x_i])$.

The smear value of a variable x_i w.r.t. a system $F(X) = 0$ in a box $[B]$ is given by the maximum impact of the variable w.r.t. each function in F, i.e.,

$$\texttt{smear}(x_i, F(X) = 0, [B]) = \max_{f \in F}(s(x_i, f, [B])).$$

The bisection strategy selects the variable with maximum smear value. A variant described in [Hansen and Walster, 2003] uses the sum of impacts.

The main drawback of this method is that some variable domains may never be split, either because their interval widths are small w.r.t. the largest interval diameter, or because their partial derivatives

have low values in the box. Anyway, these strategies can be combined in order to avoid this problem. For example, the `ALIAS` [Merlet, 2000] library has the option of combining the largest-first with the smear-based bisection methods.

This hybrid heuristic seems to have a good and more stable behaviour than the round-robin and largest-first heuristics.

3.3.2 Value selection

Once the variable has been selected, one needs to select the value at which the related interval will be split. The most used strategy consists in just bisecting (in two parts) the interval in the middle (or close to the middle). Other strategies use the information on the contraction methods for predicting goods splitting points.

There exist other strategies based on *gap detection* for splitting domains. The gaps consist of boxes without solutions inside the current box. In [Ratz, 1994] the author proposes to split variable domains inside a gap detected when applying the interval Newton method.

More recent strategies for splitting domains based on gap detection during the filtering phases have been suggested in [Benhamou et al., 1999], and studied in [Batnini et al., 2005; Chabert et al., 2005].

3.4 Other tools related to interval-based methods

In this section we describe some other techniques based on intervals for helping to solve or directly solving a system of equations. The common aspect of these three techniques is their focus on the locality problem (described in Section 3.6). They are useful to better understand two of our contributions: `I-CSE` (see Chapter 6) and `Box-k` (see Chapter 7).

3.4.1 Common subexpression elimination

CSE is a very common method in code optimization. It consists in searching identical or common subexpressions (CSs) and to analyze if is worthwhile replacing them with a single variable. Consider for example the assignment:

$$\begin{aligned} x &\leftarrow a \times b + c \\ y &\leftarrow a \times b \times c \end{aligned}$$

The computation of x and y may be achieved faster if the common subexpression $a \times b$ is replaced by an auxiliary variable tmp and the corresponding new assignment is added, i.e.:

$$\begin{aligned} tmp &\leftarrow a \times b \\ x &\leftarrow tmp + c \\ y &\leftarrow tmp \times c \end{aligned}$$

The cost/benefit analysis performed by an optimizer calculates whether the cost of storing the new variable tmp is less than the cost of the saved multiplication.

CSE generally speeds up the programs by reducing the number of instructions. Symbolic tools like Mathematica or Maple uses a DAG representation of expressions. In this representation the nodes with several parents correspond to CSs.

In interval analysis, several experts have been interested in eliminating the common subexpressions of systems of constraints. Van Hentenryck, Michel and Deville performed some replacements manually in their experimentations with Numerica [Van Hentenryck et al., 1997]. Merlet applies CSE manually before solving some systems with `ALIAS` [Merlet, 2000]. He also uses `Maple` (in the `ALIAS-Maple` variant) for rewriting systems into a DAG form. Following the DAG representation used by symbolic tools, Kearfott performs a replacement in his solver `GlobSol` [Kearfott et al., 1996]. Vu, Schichl, Neumaier and Sam-Haroud use a DAG representation of the system taking into account the CSs [Schichl and Neumaier, 2005; Vu et al., 2004, 2009b]. The community of interval analysis thought that the obtained gains were due, like in code optimization, to a reduction in the number of operations implying a faster evaluation/projection/differentiation of the functions.

In Chapter 6 we prove that, in interval constraint propagation algorithms, the gains brought by the reduction in the number of operations are negligible w.r.t. the gains due to *additional contraction*. In fact, the new system generated by CSE (represented by a DAG obtained by adding auxiliary variables) contracts better (in particular using `HC4`) than the original one due to the addition of new constraints. This explains the reported gains of several orders of magnitude on several systems.

Independently from our work, Rendl et al. [Rendl et al., 2009a,b] propose the use of CSE during *flattening*. Flattening consists of the decomposition of expressions formulated in a rich, solver-independent constraint modelling language into a conjunction of simpler expressions that conform to the constraints provided by a specific solver. The method (i.e., a flattening technique using CSE) has been tested in several discrete problems. The results and conclusions of their works are similar to ours: CSE improves the contraction power of constraint propagation algorithms (also with finite domain of variables).

3.4.1.1 A DAG representation of the system

The basic operators used by interval methods are based on a tree representation of the expression. Schichl and Neumaier, following the DAG representation used by symbolic computation tools, propose a unique directed acyclic graph (DAG) to represent the whole system of equations [Schichl and Neumaier, 2005].

The DAG representation allows us to eliminate some common subexpressions in the system. Consider the system of two equations:

$$\begin{aligned} x^2 + y + (y + x^2 + y^3 - 1)^3 + x^3 &= 2 \\ \frac{(x^2 + y^3)(x^2 + cos(y)) + 14}{x^2 + cos(y)} &= 8 \end{aligned}$$

Figure 3.7 shows an equivalent n-ary DAG of the system. Schichl and Neumaier use a binary DAG for maintaining the same unary or binary operators of evaluation and projection as with `HC4`.

The leaves of the DAG correspond to variables and constants and the roots correspond to the function expressions. The internal nodes correspond to the subexpressions. The variables and *some* subexpressions, occurring several times in the system, are represented by one node with several parents (e.g., nodes 11, 12 and 13 in the figure. For the sake of clarity each variable is represented by several nodes instead of only one).

The evaluation, the automatic projection performed by `HC4-Revise`, and the automatic differentiation

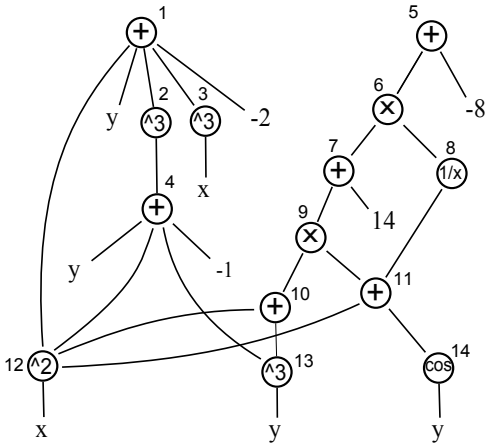

Figure 3.7: DAG representation of a system of two equations.

have been redefined on this data structure. Vu, Schichl and Sam-Haroud have described in [Vu et al., 2004, 2009b] how to carry out the propagation in the DAG. In particular, an interval is attached to internal nodes and the propagation is performed in a sophisticated way: two queues are managed, one for the evaluation, the other for the narrowing/projection, and the top-down narrowing operations have priority over the bottom-up evaluations.

The results are encouraging, the propagation in the DAG outperforms the `HC4` propagation loop by one or more orders of magnitude. The authors think that the good results are due to the reduction in the number of operations performed by CSE. However, we think that the gains are *almost only* due to the *additional contraction* allowed by CSE, as shown in Chapter 6.

3.4.2 Combining constraints

Theoretically, if we transform a system S into only one constraint c_s, then enforcing the hull-consistency in the constraint would be equivalent to enforcing global hull-consistency in the whole system, i.e., we would find the smallest box containing all the solutions of S!

Consider for example the set of two equations $C = \{x + y = 0, x - y = 0\}$. This is a very simple problem but it cannot be solved in one iteration by the `HC4-Revise` procedure due to the locality problem. We can transform C into a constraint with the same set of solutions: $c : (x + y)^2 + (x - y)^2 = 0$. Using symbolic operations we obtain $c : x^2 + y^2 = 0$. `HC4-Revise` is able to find the only solution of the constraint/system.

However, this is an exceptional case. In general, we can enforce hull-consistency only partially. If we cannot apply symbolic-based operations for reducing the number of variable occurrences in the new constraint, we cannot contract the domains further (compared to the initial form).

[Yamamura, 2000] presents a simple linear combination of nonlinear constraints. For instance from the two constraints $f(X) = 0$ and $g(X) = 0$ the redundant constraint $f(X) + g(X) = 0$ is generated. This method is useful in some quasi-linear systems (with few nonlinear subexpressions), mainly due to the elimination of common terms/variables.

[Ceberio and Granvilliers, 2002] introduces a more sophisticated strategy. The method consists of a *preprocessing* of a system of nonlinear equations of size m, where each constraint c_i is rewritten as a sum

of linear and nonlinear terms:

$$\sum_{j \in J_i} a_j x_j + \sum_{k \in K_i} b_k g_k(x_1, ..., x_n) = d_i$$

where J_i is the set of indexes related to the linear terms of c_i (a_j is the coefficient of the j^{th} linear term) and K_i is the set of indexes related to the nonlinear terms of c_i (b_k is the coefficient of the k^{th} nonlinear term $g_k(x_1, ..., x_n)$). The method consists in first replacing the nonlinear terms g_k by temporary variables u_k. Equivalent terms are replaced by the same temporary variable (like in CSE). Suppose that there are r different nonlinear terms in the original system. The obtained system is made of two subsystems, a linear one (Lc) corresponding to the original constraints:

$$Lc := \left\{ \sum_{j \in J_i} a_j x_j + \sum_{k \in K_i} b_k u_k = d_i, \quad \forall i = 1..m \right\}$$

and a nonlinear one (Ac) related to the temporary variables:

$$Ac := \{u_k = g_k(x_1, ..., x_n), \quad \forall k = 1..r\}$$

The next step consists in applying a Gaussian elimination to Lc.

The last step is the inverse of the first step, i.e., the temporary variables are replaced back by the related terms in the system Lc yielding a system L'_c. The branch and contract solving process is finally applied to L'_c.

Remark

Note that the last step contrasts with that we said about CSE in Section 3.4.1. The better results obtained by this kind of *inverse CSE* are possibly due to the use of the `Box` algorithm. Contrarily to `HC4`, applying a CSE technique in a system can lead to worse contractions of the `Box` algorithm. This problem can be avoided by using a DAG representation that maintains the values of the internal nodes during the propagation (see more details in Section 3.4.1.1). Other possible reason for the worse results is the *unknown* behavior of bisecting the new variables generated by CSE (it is difficult to predict if the bisection of the new variables improves or gets worse the results, but splitting more variables may have a bad impact on performance because it increases the size of the search space). For the moment, to avoid disastrous results due to the combinatorial explosion related to the splitting, we propose to not bisect variables generated by CSE.

Another combination of constraints is performed in actual Newton-based interval methods (described in Section 3.2.1). Recall that these methods transform the nonlinear system into a relaxed linear system using techniques from linear algebra (e.g., the Taylor extension of the equations). By using a multiplication of the Jacobian matrix by a preconditioning matrix (see Section 3.2.1.2) we obtain a linear combination of the constraints in the relaxed system.

3.4.3 The `IBB` algorithm

`IBB` (`Inter-Block Backtracking`) is an algorithm proposed by Neveu et al. [Neveu et al., 2006, 2005]. The focus of the algorithm is the solving of a system previously decomposed by a decomposition technique as `GPDOF` [Trombettoni and Wilczkowiak, 2006] or recursive rigidification.

The decomposition techniques produce a set of *well-constrained subsystems* or *blocks*. A well-constrained subsystem contains k independent equations, k variables and a finite set of solutions. All these blocks form a direct acyclic graph (DAG) as shown in Figure 3.8. Each block, represented by an ellipse, is composed by a well-constrained subsystem. The DAG indicates the order in which the blocks should be solved. For example, the block 4 requires that blocks 2 and 3 be solved before it because the solutions of these two blocks (values for variables x_2, x_3 and x_4) are used as inputs by the block 4 (i.e., constraints c_5 and c_6). Thus, the resolution algorithm should treat the subsystems respecting the order imposed by the DAG.

The `IBB` algorithm solves each block b using a bisection/contraction strategy. Each found solution is passed to each child block of b and the process is repeated until all the blocks are solved. The solutions of the leave blocks are solutions of the whole system: the values of the variables are given by the input values and the found solution. For example, if the block 4 finds the solution $x_5 = v_5$, $x_6 = v_6$ with the inputs $x_2 = p_2$,..., $x_4 = p_4$, then the related solution of the system is $X_s = (p_1, p_2, p_3, p_4, v_5, v_6)$.

The interval method used in each block is a classical contraction/bisection approach (see Algorithm 1 in Section 3.1), where the used contractors are `HC4` (or `3BCID(HC4)`) and interval Newton.

The main drawback of `IBB` is its limitation to solve decomposable systems. If the system is not decomposable, then `IBB` only applies the contraction/bisection approach to the whole system.

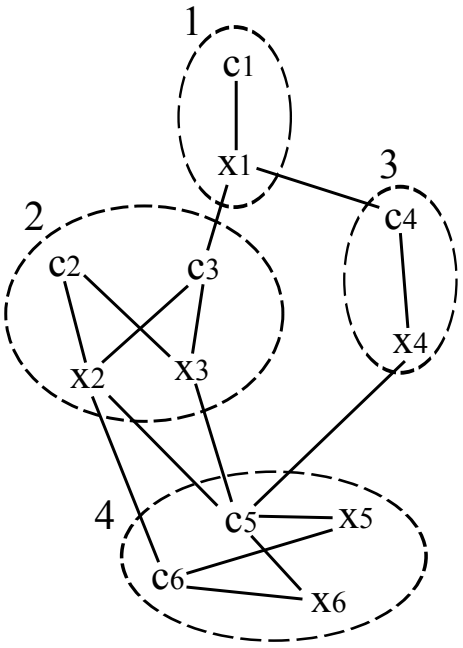

Figure 3.8: Example of DAG obtained using a decomposition technique over a system of 6 equations $(c_1, ..., c_6)$ and 6 variables $(x_1, ..., x_6)$.

The link between `IBB` and the locality problem is not so clear. To find this relation, we should view `IBB` as an algorithm that treats subsystems of equations as global constraints performing a strong consistency in each block (similar to (k+1)B-consistency in blocks of size k). This idea and the limitation of `IBB` to decomposable systems motivate us to propose a new algorithm for generalizing the method to sparse non-decomposable systems (see Chapter 7).

3.5 Interval-based solving tools

There exist several interval-based solvers or libraries for solving problems using the intervals. We briefly introduce some of the most used, influential and/or promising tools.

- `Numerica` (presented in [Van Hentenryck et al., 1997]) is one of the first autonomous tools proposed by P. Van Hentenryck, L. Michel and Y. Deville. It allowed minimizing nonlinear objective functions with or without constraints. It is not supported anymore and can be found in a `C++` library of Ilog Solver (IlcNum). The authors of `Numerica` intend to integrate a new interval library in the tool/language `Comet` ([Van Hentenryck and Michel, 2005]). `Comet` is currently dedicated to local search and finite domain CSPs.
- `RealPaver` ([Granvilliers and Benhamou, 2006]) was designed in the nineties. It is specialized in the resolution of systems of equations using interval methods. Since 2008 it is available as a `C++` library.
- `ALIAS` ([Merlet, 2000]) is the tool used by the COPRIN team for robotic applications. It is oriented to the resolution of systems of equations using operators from interval analysis and constraint programming. The `Maple-ALIAS` version is the first solver integrating interval-based methods and techniques of symbolic computation.
- `Icos` ([Lebbah et al., 2005]) is specialized in robust constrained global optimization. The library contains algorithms coming from constraint programming, interval analysis, and linear relaxation techniques. It also contains an interface to some linear programming and local optimization solvers (e.g., `Coin/Clp`, `Cplex`, and `IpOpt`).
- `GloptLab`, developed in `Matlab` by Ferenc Domes ([Domes, 2009]), is specialized in quadratic systems. It implements several contraction algorithms: scaling, constraint propagation, linear relaxations, strictly convex enclosures and conic methods.
- `Ibex` ([Chabert, 2009]) is the library used for implementing the algorithms proposed in this thesis. `Ibex` has been developed by Gilles Chabert. For the moment, it is limited to the resolution of systems of equations. Since 2008 a new high-level language (`Quimper` [Chabert and Jaulin, 2009c]) has been implemented on the top of `Ibex`, allowing non-specialist engineers to use the library more easily.

3.6 Other research fields related to intervals

Interval-based methods are not only used for solving systems of equations. Several researchers also use these methods for solving global optimization problems and finding continuous solutions of under-constrained systems (*paving*).

Global optimization

The global optimization problem consists in minimizing a nonlinear objective function subject to nonlinear equations and inequalities.

Classical approaches based on linear relaxations and local methods are the most successful ones due to their efficiency. The most famous implementation is the global optimizer called `Baron`. However, these

methods are not *rigorously implemented*. That is, the result of these algorithms could be an overestimation or, even worse, an underestimation of a global optimum.

The interest in interval methods is mainly due to the safe bounding of the global optimum. For instance, the optimization framework `QuadSolver` [Lebbah et al., 2005] uses safe linear relaxations to reduce the domains of the variables. Linear relaxations are combined with local consistencies as well as interval methods (e.g., interval Newton) to provide an efficient and safe framework to obtain the solutions of nonlinear optimization problems.

Continuous solution sets

Several interval-based approaches have been proposed for treating systems with infinite sets of solutions. The classical approach consists in extracting from this set one solution minimizing a given criterion (global optimization).

For representing the continuous solutions in a compact way, an alternative is to work with *inner* and *outer* boxes. The outer boxes are the boxes that do not contain any solution. The standard interval-based contractors described in Section 3.2 (e.g., `HC4`, `3B`) can guarantee the absence of solutions in a given box. An inner box is a box $[B]$ where all the elements are solutions of the system. Consider the system: $\{F(X) = 0, G(X) \leq 0, H(X) \geq 0\}$. $[B]$ is an inner box if:

$$\forall V \in [B]: \quad F(V) = 0, G(V) \leq 0, H(V) \geq 0$$

In other words, an inner box is contained inside the solution set of the system. Inner boxes are not bisected and are immediately considered as a solution of the system. Thus, effective (and efficient) *inner tests* are required for improving the performance of the solver.

An inner test, (that is, the test for proving that a box is in fact an inner box), for a system made of inequalities $C : \{c_1, ..., c_n\}$ must satisfy the relation:

$$\texttt{Sat}\forall?(c_1, [B]) \wedge ... \wedge \texttt{Sat}\forall?(c_n, [B]) \Leftrightarrow \textbf{true}$$

where $\texttt{Sat}\forall?(c_i, [B])$ is true if the constraint c_i is satisfied for every element in $[B]$. The simplest way to prove that a box is inside the solution set is to transform the relation using a double negation:

$$\texttt{Sat}\exists?(\neg c_1, [B]) \vee ... \vee \texttt{Sat}\exists?(\neg c_n, [B]) \Leftrightarrow \textbf{false}$$

and to prove the unsatisfiability of each negative constraint. ($\texttt{Sat}\exists?(c_i, [B])$ is false if every element in $[B]$ does not satisfy c_i.) Tests based on this idea are used in several applications developed by Luc Jaulin [Jaulin et al., 2001] and JP Merlet [Merlet, 2007]. It has also been formalized in [Rueher et al., 2008] and [Benhamou and Goualard, 2004] and implemented in different solvers (e.g., `Ibex` [Chabert, 2009; Chabert and Jaulin, 2009a]).

Only continuous solution sets in low dimension (3 or less) can be inner and outer sharply approximated by interval-based methods.

In [Sam-Haroud and Faltings, 1996], the authors propose to use 2^k trees for representing regions of feasible solutions related to k variables. They also propose strong consistency algorithms, that terminate in polynomial time, to reduce the search space using this kind of representation.

3.7 Conclusion

In the present chapter we have described several interval-based techniques used for solving systems of constraints. They are mainly based on a branch & prune method performing a tree search. In each node one or more contractors reduce the domains of the variables. We distinguish three kinds of contractors: contractors based on the interval Newton method, treating all the system as one global constraint; linear relaxation contractors using linear inequalities for approximating the nonlinear constraints; and constraint programming contractors, including constraint propagation algorithms.

Our main contributions are focused on the last category. The basic constraint propagation algorithms try to enforce the hull-consistency in the system, i.e., to find the smallest box satisfying each constraint independently (similar to enforce arc-consistency in finite domain CSPs). The algorithms perform a revise procedure consisting in projecting each constraint over each of its variables implying a reduction of the domains. Then, the reductions are propagated to the rest of the system. Algorithms enforcing stronger consistencies increase the scope of the basic constraint propagation algorithms and allow better reductions.

The most important limitations of the constraint propagation algorithms, and in general any other interval-based method, are due to the *dependency problem* and to *the locality problem*. The dependency problem is the main cause of non-optimal evaluations of functions but also, analogously, the main cause of non-optimal projections of functions (approximating hull-consistency).

Just as the dependency problem is the main reason for which we cannot enforce (local) hull-consistency over a constraint, the locality problem is the main reason for which we cannot enforce *global* hull-consistency over the whole system (i.e., finding the smallest box containing all the solutions of the system). The locality problem is mainly caused by the locality scope of filtering algorithms.

In Part 2 of this thesis we present our contributions mainly related with the dependency and the locality problems.

Part II

Contributions

Introduction

In the second part of this thesis we present our contributions. The first two of them are related to the dependency problem.

In Chapter 4 we propose a new constraint propagation algorithm (`Mohc`) for better approximating hull-consistency. The algorithm is based on the two classical procedures `HC4-Revise` and `Box-Revise` that have been adapted for using monotonicity. Our procedure `MinMaxRevise` (based on `HC4-Revise`) allows us to project *using the monotonicity* on variables that are not monotonic and those that appear once in the function (a further version `MinMaxRevise'` allows us to project on all the variables). Our procedure `MonotonicBoxNarrow` (based on `Box-Revise`) reduces the domain of each monotonic variable[1] appearing several times in the function. While the classic `Box-Revise` algorithm has a worst-case time complexity of $O(1/\epsilon)$ (where ϵ is the required precision), our algorithm performs a dichotomic process having a time complexity of $O(log(1/\epsilon))$. `Mohc` is able to enforce the hull-consistency in functions that are monotonic w.r.t. all their variables appearing several times.

In Chapter 5 we present Occurrence Grouping (OG), a new interval extension based on monotonicity and used by `Mohc`. OG transforms each non-monotonic variable into a convex sum of three auxiliary variables, two of them being monotonic. The new expression computes the same natural evaluation as the original one but a sharper evaluation by monotonicity.

The following two contributions are related to the locality problem.

In Chapter 6 we prove that eliminating common subexpressions in a system (see Section 3.4.1) improves the filtering of constraint propagation algorithms, in particular `HC4`. Then we propose `I-CSE`, an algorithm able to find *all* the *maximal* common subexpressions shared by each pair of equations. `I-CSE` replace them by auxiliary variables. When two or more CSs are in conflict, `I-CSE` adds the necessary redundant equations to solve it.

The second contribution related to the locality problem is described in Chapter 7. We propose a new algorithm for enforcing strong consistencies using well-constrained subsystems. The idea is inspired from the `IBB` algorithm (for solving decomposable systems) but it is extended to systems that are sparse while irreducible.

[1] In the sake of conciseness, when a function is monotonic w.r.t. a variable x, we say that x is *monotonic*, otherwise we say that x is *non-monotonic*.

Chapter 4

An Algorithm Exploiting Monotonicity

Contents

In this chapter we present `Mohc-Revise`, a new revise procedure that can be used in a constraint propagation algorithm (see Section 3.2.3). The algorithm reduces the dependency problem of a constraint $f(X) = 0$ when f is monotonic w.r.t. one or several variables with multiple occurrences[1].

When f is monotonic w.r.t. *all* the variables with multiple occurrences (and f is continuous and differentiable), `Mohc-Revise` is proven to compute the hull-consistency, i.e., it returns the smallest box enclosing all the solutions of the constraint.

(A part of the material presented in this chapter is published in [Araya et al., 2009b].)

4.1 Introduction

Recall that the monotonicity properties of functions are used in several interesting ways by interval methods. For instance, the basic operators of interval arithmetic ($+$, $-$, $\times$, $/$, sin, log, ...) use these properties for computing optimal images of atomic expressions (see more details in Section 2.2). The extension by monotonicity also uses these properties for computing sharper evaluation of more complicated functions (see Section 2.4.2).

The `ALIAS` library [Merlet, 2000], the `Octum` algorithm [Chabert and Jaulin, 2009b] and the proposal described in [Goldsztejn et al., 2009] are three examples of how we can use the monotonicity for contracting domains. Goldsztejn et al. use monotonicity of functions in quantified NCSPs to easily contract

[1]Thanks to the occurrence grouping extension (see Section 4.3.2 and Chapter 5) it is enough that f is monotonic w.r.t. some occurrences of a single variable for improving the contraction.

a universally quantified variable that is monotonic. The method used by ALIAS (when the related option is activated) contracts the domain of each occurrence o involved in a constraint $f(X) = 0$ by evaluating the *projection function* of o using the extension by monotonicity. The projection function of an occurrence o is obtained by isolating o in the constraint $f(X) = 0$ (see Section 3.2.3.2). (In addition, contrarily to HC4-Revise or Mohc-Revise, ALIAS generates the projection functions *symbolically* before using them.) The constraint propagation algorithm Octum is briefly described in Section 3.2.3.6 and is similar to one of the main procedures in Mohc-Revise (see Section 4.2.2).

Due to the *dependency problem* (see Section 2.3.3) when a variable occurs more than once in an equation, the evaluation and contraction performed by the HC4-Revise algorithm are generally not optimal (see Section 3.2.3.4), i.e., the algorithm cannot compute the hull-consistency. In equations with *only* one variable x occurring several times BoxNarrow can project optimally over x. However, if several variables occur several times, there is no efficient technique to compute the optimal projections of all the variables.

When a function f is monotonic w.r.t. a variable x in a given box, it is well-known that the monotonicity-based interval extension of f produces no overestimation related to the multiple occurrences of x (see Section 2.4.2). However, this property has not been exploited for computing optimal projections.

4.2 The MOnotonic Hull Consistency algorithm

The MOnotonic Hull Consistency algorithm (in short Mohc) is a new constraint propagation algorithm that exploits monotonicity of functions to better contract a box. The propagation loop is exactly the same AC3-like algorithm performed by the famous HC4 and BC3 (see Section 3.2.3). Its novelty lies in the Mohc-Revise procedure handling one constraint individually and described in Algorithm 6.

Algorithm 6 Mohc-Revise (**in-out** $[B]$; **in** f, Y, W, ρ_{mohc}, τ_{mohc}, ϵ)

```
HC4-Revise(f(Y,W) = 0, Y, W, [B])
if W ≠ ∅ and ρ_mohc[f] < τ_mohc then
  ([G], [G_o]) ← GradientCalculation(f, W, [B])
  (f^og, W) ← OccurrenceGrouping(f, W, [B], [G_o])
  (f_max, f_min, X, W) ← ExtractMonotonicVars(f^og, W, [B], [G])
  MinMaxRevise([B], f_max, f_min, Y, W)
  MonotonicBoxNarrow([B], f_max, f_min, X, [G], ϵ)
end if
```

This procedure aims at narrowing the current box $[B]$. It works on a unique equation[1] $f(Y, W) = 0$, in which the variables in Y occur once in the expression f whereas the variables in W occur several times in f.

Mohc-Revise starts by a call to HC4-Revise (an exception terminating the procedure is raised if an empty box is obtained, proving the absence of solution). If f contains variables with multiple occurrences ($W \neq \emptyset$) and if another condition is fulfilled (see Section 4.3.1), then five procedures are called to detect and exploit the monotonicity of f.

The GradientCalculation function computes the gradient of f. More precisely, for each variable w in W it computes the interval partial derivative of f w.r.t. w and the interval partial derivative of f w.r.t. *each occurrence* of w in the box $[B]$ resulting in interval vectors $[G]$ and $[G_o]$ respectively. The

[1] The procedure can be straightforwardly extended to handle an inequality.

interval gradient is computed using the backward automatic differentiation method (AD) described in Section 2.2.4.2.

The `OccurrenceGrouping` function is not required in `Mohc-Revise` and can be viewed as an improvement of it. It rewrites the expression f into a new form f^{og} such that the image $[f^{og}]_M([B])$ computed by the monotonicity-based interval extension is sharper than, or at worst equal to, the image $[f]_M([B])$. This sophisticated function is briefly introduced in Section 4.3.2 and deeply explained in Chapter 5.

The `ExtractMonotonicVars` procedure uses the vector of interval partial derivatives $[G]$ for finding the monotonic variables in W and to move them to X. For every variable $w_i \in W$ the procedure checks whether $0 \in [g_i]$, where $[g_i] \in [G]$ is the interval partial derivative related to w_i in the box (i.e., $[g_i] = \left[\frac{\partial f}{\partial w_i}\right]([B])$). If it does not, it means that f^{og} is monotonic w.r.t. w_i, so that w_i is removed from W to be added into X. At the end:

- X contains variables with multiple occurrences that are monotonic.
- W contains variables that are not detected to be monotonic.
- Y contains variables that appear only once in the function.

In the following, $[B] = [X] \times [Y] \times [W]$. Consider a variable $x_i \in X$ ($X = (x_1, ..., x_n)$). We denote $x_i^- \in [x_i]$ the value of the variable x_i that minimizes f^{og} for any combination of the other elements in $[B]$. Thus, if f^{og} is increasing (resp. decreasing) w.r.t. x_i, then $x_i^- = \underline{x_i}$ (resp. $x_i^- = \overline{x_i}$). Symmetrically, x_i^+ is the value of x_i that maximizes f in the box. We also define two real vectors $X^+ = (x_1^+, ..., x_n^+)$ and $X^- = (x_1^-, ..., x_n^-)$.

The `ExtractMonotonicVars` procedure generates two *real* functions f_{min} and f_{max}:

$$\begin{aligned} f_{min}(Y, W) &= f^{og}(X^-, Y, W) \\ f_{max}(Y, W) &= f^{og}(X^+, Y, W) \end{aligned}$$

Observe that the natural extension of these functions provides the bounds of the evaluation by monotonicity (see Definition 3 in Section 2.4.2), i.e.,

$$[f^{og}]_M([B]) = \left[\underline{[f_{min}]([Y], [W])}, \overline{[f_{max}]([Y], [W])}\right]$$

The next two routines are in the heart of `Mohc-Revise` and are detailed below. They mainly work with the two functions f_{min} and f_{max}. The procedure `MinMaxRevise` narrows the variables in Y (appearing once in f and thus in f^{og}) and those in W. The procedure `MonotonicBoxNarrow` narrows the monotonic variables in X.

At the end, if `Mohc-Revise` has contracted the interval of a variable in W (more than a user-defined ratio τ_{propag}), then the constraint is pushed into the propagation queue in order to be handled again in a subsequent call to `Mohc-Revise`. Otherwise, we know that a fixpoint in terms of filtering has been reached (under some assumptions). Indeed, nice properties presented in Section 7.3.5 explain why, when $W = \emptyset$, `MinMaxRevise` contracts $[Y]$ optimally while `MonotonicBoxNarrow` contracts $[X]$ optimally. The constraint is thus not pushed into the propagation queue.

Note that a variable y in Y, appearing once in f, is handled by `MinMaxRevise`, and not by `MonotonicBoxNarrow`, *even if it is monotonic*. We have made this choice because `MinMaxRevise` is less costly than `MonotonicBoxNarrow` (see Proposition 12, page 84) and has *the same filtering power on* $[y]$ (see Lemma 4, page 84).

4.2.1 The MinMaxRevise procedure

Algorithm 7 MinMaxRevise (**in-out** $[B]$; **in** f_{max}, f_{min}, Y, W)

HC4-Revise($f_{min}(Y,W) \leq 0, Y, W, [B]$) /* also called MinRevise */
HC4-Revise($f_{max}(Y,W) \geq 0, Y, W, [B]$) /* also called MaxRevise */

MinMaxRevise brings a contraction on variables in Y and W.

Since

$$(\exists X \in [X])(\exists Y \in [Y])(\exists W \in [W]): \quad f(X,Y,W) = 0 \Rightarrow f_{min}(Y,W) \leq 0 \text{ and } 0 \leq f_{max}(Y,W)$$

the MinMaxRevise contractor for the constraint $f(X,Y,W) = 0$ is obtained calling HC4-Revise on the constraints $f_{min}(Y,W) \leq 0$ and $0 \leq f_{max}(Y,W)$.

Thus, MinMaxRevise narrows intervals of variables in Y and W using the monotonicity of variables in X.

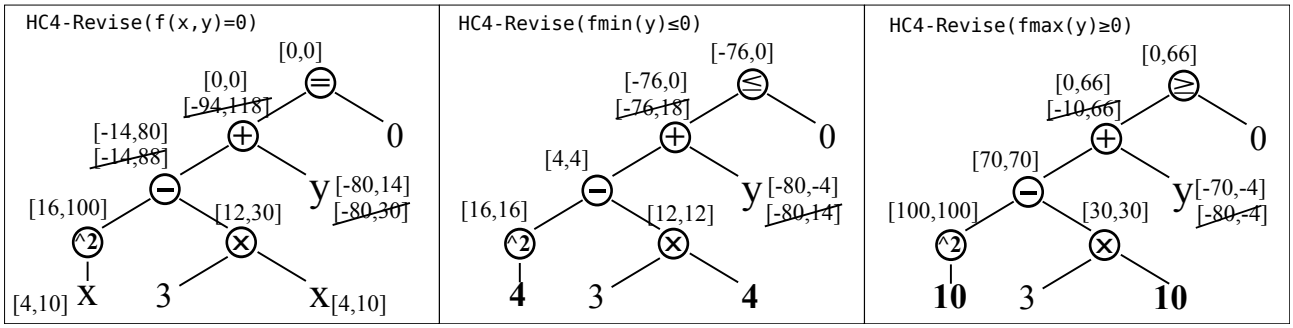

Figure 4.1: HC4-Revise (**Left**), MinRevise (**Center**) and MaxRevise (**Right**) applied to $x^2 - 3\,x + y = 0$.

Figure 4.1 illustrates how the first part of Mohc-Revise narrows the box $[x] \times [y] = [4,10] \times [-80,30]$ of the constraint: $f(x,y) = x^2 - 3\,x + y = 0$.

HC4-Revise works in two phases (Figure 4.1-left). The evaluation phase evaluates every node bottom-up and attaches the result to the node. The second phase, due to the equality node, starts by intersecting the top interval $[-94,118]$ with 0, and proceeds top-down by applying the narrowing operators (more details about the HC4-Revise algorithm and the narrowing operators are provided in Section 3.2.3.4). After this step, OccurrenceGrouping detects that the function is monotonic w.r.t. x (trivial case when the derivative is positive so that $f = f^{og}$), hence ExtractMonotonicVars puts x in the set X of monotonic variables.

Figure 4.1-center shows the first step of MinMaxRevise. The tree represents the inequality $f_{min}(y) = f(4,y) \leq 0$. Calling HC4-Revise on this expression produces a new contraction on y because x is replaced by $x^- = \underline{x} = 4$. On the top of the tree, the narrowing phase intersects $[-76,18]$ with $[-\infty,0]$ (inequality), and the first narrowing operator yields $[y] \leftarrow [-80,14] \cap ([-76,0] - [4,4]) = [-80,-4]$. Following the same principle, MaxRevise applies HC4-Revise to $f_{max}(y) = f(10,y) \geq 0$ and narrows $[y]$ to $[-70,-4]$ (see Figure 4.1-right).

Note that MinMaxRevise does not contract the monotonic variables in X. These variables are contracted by the MonotonicBoxNarrow procedure.

4.2.2 The `MonotonicBoxNarrow` procedure

The procedure `MonotonicBoxNarrow` has similarities with the `BoxNarrow` algorithm described in Section 3.2.3.5 and the `Octum` algorithm described in Section 3.2.3.6. It aims at narrowing the interval of every *monotonic variable* x_i in X. For performing the narrowing of x_i, `MonotonicBoxNarrow` works with *two interval* functions:

$$[f^{x_i}_{min}](x_i) = [f^{og}]_n(x_1^-, ..., x_{i-i}^-, x_i, x_{i+1}^-, ..., x_{n'}^-, [Y], [W]) \quad (4.1)$$

$$[f^{x_i}_{max}](x_i) = [f^{og}]_n(x_1^+, ..., x_{i-i}^+, x_i, x_{i+1}^+, ..., x_{n'}^+, [Y], [W]) \quad (4.2)$$

Observe that all the variables in X, excepting x_i, have been replaced by one bound of the related interval. The variables in Y and W have been replaced by their domains. $[f^{x_i}_{max}]$ and $[f^{x_i}_{min}]$ denote univariate thick/interval functions depending on x_i (see Figure 4.3).

`MonotonicBoxNarrow` calls two subprocedures:

- If x_i is increasing it calls `LeftNarrowFmax` and `RightNarrowFmin`.
- If x_i is decreasing it calls `LeftNarrowFmin` and `RightNarrowFmax`.

W.l.o.g. consider in the following that f is increasing. The `LeftNarrowFmax` procedure attempts to find the leftmost zero of the function $[f^{x_i}_{max}]$ in the interval $[x_i]$ for improving the left bound of the interval. In the same way, `RightNarrowFmin` attempts to find the rightmost zero of the function $[f^{x_i}_{min}]$ and narrows the right bound of $[x_i]$.

We detail below how the left bound of $[x_i]$ is improved by the `LeftNarrowFmax` procedure. Note that if `MonotonicBoxNarrow` is called, this required `MinMaxRevise` be terminated with no failure. This means that `LeftNarrowFmax` will never return an empty interval for $[x_i]$: either $[x_i]$ is left unchanged, or $[x_i]$ is narrowed to a non-empty interval (see Lemma 1 in Section 4.4.1).

4.2.3 The `LeftNarrowFmax` procedure

This procedure has a close connection with the `LeftNarrow` procedure used by the well-known `Box` algorithm (see Section 3.2.3.5 or [Benhamou et al., 1999; Van Hentenryck et al., 1997]). However, because f is monotonic w.r.t. x_i, the contraction process is faster, that is, it is a true dichotomic, and thus log-time, process.

Algorithm 8 `LeftNarrowFmax` (**in-out** $[x]$; **in** $[f^x_{max}]$, $[g]$, ϵ)

if $\overline{[f^x_{max}](\underline{x})} < 0$ /* test of existence */ **then**
  $size \leftarrow \epsilon \times$ `Diam`$([x])$
  $[l] \leftarrow [x]$
  while `Diam`$([l]) > size$ **do**
    $x_m \leftarrow$ `Mid`$([l])$; $z_m \leftarrow \overline{[f^x_{max}](x_m)}$ /* $z_m \leftarrow \underline{[f^x_{min}](x_m)}$ in {`Left|Right`}`NarrowFmin` */
    $[l] \leftarrow [l] \cap x_m - \frac{z_m}{[g]}$ /* Newton iteration */
  end while
  $[x] \leftarrow [\underline{l}, \overline{x}]$
end if

Let us illustrate `LeftNarrowFmax` (Algorithm 8) applied to the $[f^x_{max}]$ function depicted in Figure 4.2. Starting with $[l_0] = [x]$ (the index corresponds to the iteration number), the goal is to contract $[l]$ for providing a tight approximation of the point L, i.e., the new left bound of $[x]$. `LeftNarrowFmax` provides a sharp enclosure of L and keeps only its left bound at the end (last line of Algorithm 8 and step 4 on the figure).

A first existence test checks that $\overline{[f^x_{max}](\underline{x})} < 0$, i.e., the point A in Figure 4.2-left is below zero. Otherwise, $[f^x_{max}] \geq 0$ is satisfied in $\underline{x}$ so that $[x]$ cannot be narrowed, leading to an early termination of the procedure.

A dichotomic process is then run until `Diam`$([l]) \leq size$. A classical Newton iteration is iteratively launched from the midpoint x_m of $[l]$, e.g., from the point B (middle of $[l_0]$) and from the point C (middle of $[l_1]$) in the figure.

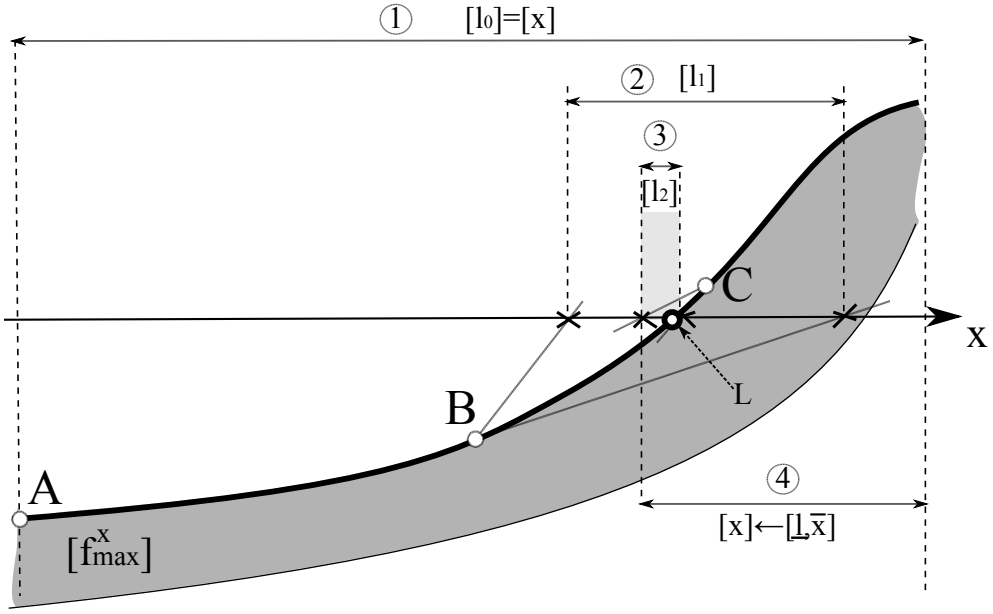

Figure 4.2: Interval Newton iterations for narrowing the left bound of $[x]$.

Graphically, an iteration of the univariate interval Newton (described in Section 3.2.1) intersects $[l]$ with the projection on the x axis of the cone (two lines emerging from B and C) enclosing the function. The slopes of the lines bounding the cone are equal to the bounds of the interval partial derivative of x. Note that the cones forms an angle of at most 90 degrees because the function is monotonic and the interval partial derivative $[g]$ is positive[1]. This explains why the diameter of $[l]$ is divided by at least 2 at each iteration. Indeed, if $z_m < 0$ (point B in Figure 4.2), then the term $-\frac{z_m}{[g]}$ is positive and the dichotomic process will continue in the right side of x_m (step 2). If $z_m > 0$ (point C), then $-\frac{z_m}{[g]} < 0$ and one will proceed with the left side of x_m (step 3). (The following property can also be obtained if we remove the Newton iteration from the procedure.)

Proposition 9 *The procedures* `LeftNarrowFmax`*,* `RightNarrowFmin`*,* `LeftNarrowFmin`*,* `RightNarrowFmax` *terminate and run in time* $O(log(\frac{1}{\epsilon}))$*, where* ϵ *is a precision expressed as a ratio of interval diameter.*

At the end, the procedure computes a new left bound for $[x]$ (step 4).

[1]Every Newton iteration could recompute a more narrow interval derivative $[g]$ (implying a thinner cone) although we do not perform it in our implementation.

Observe that Newton iterations called inside `LeftNarrowFmax` and `RightNarrowFmax` work with $z_m = \overline{[f^x_{max}](x_m)}$, that is, a *degenerate* curve (in bold in the figure), and not with a thick function. In the same way, `LeftNarrowFmin` and `RightNarrowFmin` work with $z_m = \underline{[f^x_{min}](x_m)}$. As z_m is a floating point number, it overestimates the real evaluation $\overline{[f^x_{max}](x_m)}$ (or underestimates the evaluation of $\underline{[f^x_{min}](x_m)}$). For this reason it is possible, in some cases, to lose *the unique solution.* In Section A.8 we propose a modification for avoiding this problem.

4.3 Advanced features of Mohc-Revise

In this section we describe some advanced features of our algorithm.

4.3.1 The user-defined parameter τ_{mohc} and the array ρ_{mohc}

The user-defined parameter $\tau_{mohc} \in [0, 1]$ allows the monotonicity-based procedures to be called more or less often (see Algorithm 6). For every constraint, $\rho_{mohc}[f]$ tries to predict whether the monotonicity-based treatment that follows is promising: the procedures exploiting monotonicity are called only if $\rho_{mohc}[f] < \tau_{mohc}$. These ratios are computed in a preprocessing procedure called after every bisection (i.e., choice point) on the current box $[B]$, as follows[1]:

$$\rho_{mohc}[f] = \frac{\texttt{Diam}([f]_M([B]))}{Diam([f]([B]))} = \frac{\texttt{Diam}\left(\left[\underline{[f_{min}]([B])}, \overline{[f_{max}]([B])}\right]\right)}{\texttt{Diam}([f]([B]))}$$

This ratio thus indicates whether the monotonicity-based image of a function is sufficiently sharper than the natural one. As confirmed by the experiments detailed in Section 4.6, this ratio is relevant for the bottom-up evaluation phases of `MinRevise` and `MaxRevise`, and also for `MonotonicBoxNarrow`.

The experiments, reported in Section4.6.2, show that the parameter τ_{mohc} is useful when `Mohc` is a subcontractor of `3BCID` [Trombettoni and Chabert, 2007]. In this case, `Mohc` is called many times between two branching points, so that the CPU time required by the preprocessing procedure filling the array ρ_{mohc} is negligible. This trend would also be true for any other sophisticated contractor calling a constraint propagation subcontractor, such as `3B` [Lhomme, 1993], `Quad` [Lebbah et al., 2005] or `Box-k` (described in Chapter 7). When `Mohc` is called only once between two branching points, the parameter τ_{mohc} seems to be less effective.

4.3.2 The `OccurrenceGrouping` procedure

(See Chapter 5 for more details.) The `OccurrenceGrouping` procedure rewrites the expression f into a new form f^{og} such that the image $[f^{og}]_M([B])$ computed by the monotonicity-based interval extension is sharper than, or at worst equal to, the image $[f]_M([B]) = [f]([B])$.

For example, consider the function $f_1(x) = -x^3 + 2x^2 + 6x$, with $[x] = [-1.2, 1]$. f_1 is not detected monotonic since the image of $[-1.2, 1]$ by natural evaluation of the derivative $f_1'(x) = -3x^2 + 4x + 6$

[1] To compute $\rho_{mohc}[f]$, the preprocessing procedure calls almost the same procedures as `Mohc-Revise`, i.e., `GradientCalculation` (computing $[f]([B])$), `OccurrenceGrouping`, `ExtractMonotonicVars` and the evaluation phase of `MinMaxRevise` (computing $[f_{min}]([B])$ and $[f_{max}]([B])$).

contains 0. `OccurrenceGrouping` produces:

$$f_1(x) = f_1^{og}(x_a, x) = -x_a^3 + 2(0.35x_a + 0.65x)^2 + 6\,x_a$$

where x_a is an increasing variable in the new function f_1^{og}. Then, we can observe that:

$$[f_1^{og}]_M([x]) = [-5.472, 7] \subseteq [f_1]_M([x]) = [-8.2, 10.608]$$

At the end, `OccurrenceGrouping` returns the new expression f^{og} and a new set W that includes the new variables (e.g., x_a), if any. (`ExtractMonotonicVars` transfers x_a into the set X of monotonic variables.)

To maintain the equivalence between the initial expression and the new one, we should add the constraint relating the new auxiliary variables with x (i.e., $x = x_a$). However, as f^{og} lives only inside the `Mohc-Revise` procedure, we prefer to evaluate x_a using the interval $[x]$ (i.e., $[x_a] \leftarrow [x]$) and to update the interval $[x]$ immediately after a contraction of $[x_a]$, i.e., $[x] \leftarrow [x] \cap [x_a]$.

The `OccurrenceGrouping` procedure has a time complexity of $O(k\,log_2(k))$ for each variable occurring k times in a function f (see Section 5.5, page **??**).

4.4 Understanding and improving `Mohc-Revise`

The aim of this section is to better understand the process performed by `Mohc-Revise`, especially by its two most interesting procedures: `MinMaxRevise` and `MonotonicBoxNarrow`. Then, we propose improvements that are detailed in Section 4.4.4.

4.4.1 `MinMaxRevise` ensures the existence of a solution in the box

Lemma 1 *Consider a function f continuous in the box $[B] = [X] \times [Y] \times [W]$. When* `MinMaxRevise` *is successfully applied in f for contracting $[B]$, it certifies that there exists a vector $V \in [X]$ that satisfies $0 \in [f]_M(V, [Y], [W])$.*

Proof 5 *After having successfully called the* `MinMaxRevise` *procedure, the following conditions are satisfied:*

$$\overline{[f_{max}]([Y], [W])} = \overline{[f](X^+, [Y], [W])} \geq 0$$
$$\underline{[f_{min}]([Y], [W])} = \underline{[f](X^-, [Y], [W])} \leq 0$$

There exist two cases:

- *If $[f_{min}]([Y], [W])$ (resp. $[f_{max}]([Y], [W])$) contains 0, then the vector $V = X^-$ (resp. $V = X^+$) satisfies the relation $0 \in [f]_M(V, [Y], [W])$.*
- *If $[f_{min}]([Y], [W]) < 0$ and $[f_{max}]([Y], [W]) > 0$, i.e., $\underline{[f]_{opt}([B])} \subseteq [f_{min}]([Y], [W]) < 0$ and $\overline{[f]_{opt}([B])} \subseteq [f_{max}]([Y], [W]) > 0$, then $0 \in [f]_{opt}([B])$.*
  As f is continuous in $[B]$, $0 \in [f]_{opt}([X], [Y], [W])$ implies that there exists a vector $V \in [X]$ such that $0 \in [f]_{opt}([X], [Y], [W]) \subseteq [f]_M(V, [Y], [W])$. □

Thus, Lemma 1 implies that `MonotonicBoxNarrow` called after `MinMaxRevise` cannot result in an empty box.

4.4.2 Duality of the contraction process

During MinMaxRevise, every $x_i \in X$ is replaced by one of its bounds $\overline{x_i}$ or $\underline{x_i}$. These bounds could been modified by the next call to MonotonicBoxNarrow, forcing to repeat the process. Fortunately, the following lemma shows that iterating over these two methods does not improve the contraction.

Lemma 2 *If after applying* MinMaxRevise *and* MonotonicBoxNarrow *to a box* $[B] = [X] \times [Y] \times [W]$, $[W]$ *is not contracted, then a second call to* MinMaxRevise *or* MonotonicBoxNarrow *cannot further contract* $[B]$.

Proof 6 *Let us detail this point on* T_{fmax} *(the same reasoning holds for* T_{fmin}*), the expression tree representing* f_{max}*, and with an increasing variable* x_i*. Before the call to* MaxRevise*, since* x_i *is increasing, it is replaced by* $\overline{x_i}$ *in* T_{fmax}*. During* MaxRevise*, at the end of the bottom-up evaluation phase of* T_{fmax}*, the root interval is* $[z] = [f_{max}]([Y], [W]) = [\underline{z}, \overline{z}]$*. Three cases may occur according to the signs of* $\underline{z}$ *and* $\overline{z}$*. Figure 4.3 illustrates the two interesting cases.*

The first case (case A in Figure 4.3) occurs when $\underline{z} \leq 0 \leq \overline{z}$*, resulting in* $[z'] = [0, \overline{z}]$ *after intersection of* $[z]$ *with* $[0, +\infty]$ *in the top of* T_{fmax} *(because* $[f_{max}]([Y], [W]) \geq 0$*), and thus in a potential contraction of* $[y_j]$*. However,* MonotonicBoxNarrow *cannot contract the right bound of* $[x_i]$ *(using* $[f^{x_i}_{min}]$*) since* $\overline{x_i}$ *is already a zero (*$\overline{x_i} \in [L, R]$ *in the figure). This implies that the* RightNarrowFmin *procedure of* MonotonicBoxNarrow *(resp. the* LeftNarrowFmin *procedure for decreasing variables) is useless if* $\underline{z} = \underline{[f_{max}]([Y], [W])} \leq 0$*. The situation is symmetric if* $\overline{[f_{min}]([Y], [W])} \geq 0$*.*

The second case (case B in Figure 4.3) occurs when $0 < \underline{z} < \overline{z}$*.* $[z]$ *does not change after being intersected with* $[0, +\infty]$*. Since* $[z]$ *is not narrowed, no* $[y_j]$ *(with* $y_j \in Y$*) is contracted in the top-down narrowing phase of* T_{fmax}*. Next,* MonotonicBoxNarrow *reduces* $\overline{x_i}$ *to* $\overline{x'_i}$*, but* $0 < \underline{z}' < \overline{z}'$ *remains true (see figure). Thus, a second call to* MaxRevise *could not further contract* $[y_j]$*.*

A last trivial case occurs when $\underline{z} < \overline{z} < 0$*.* MaxRevise *detects that there is no solution due to an empty intersection with* $[0, +\infty]$*.* □

Cases A and B of the proof highlight the duality of the contraction process. This duality explains why we do not push the revised constraint again into the propagation queue when there is no reduction of $[W]$.

4.4.3 When the narrowing procedures are useless

Lemma 3 *If* MonotonicBoxNarrow *reduces the interval of a variable* $x_i \in X$ *using* $[f^{x_i}_{max}]$ *(resp.* $[f^{x_i}_{min}]$*) or, if* $0 \in [f^{x_i}_{max}](x_i^-)$ *(resp.* $0 \in [f^{x_i}_{min}](x_i^+)$*), then, for all* $j \neq i$*,* $[f^{x_j}_{min}]$ *(resp.* $[f^{x_j}_{max}]$*) cannot bring any additional narrowing to* $[x_j]$*.*

Proof 7 *With no loss of generality, let us explain the principle assuming that* f *is increasing w.r.t.* x_i*. Since* x_i *is increasing* LeftNarrowFmax *is called to narrow the left bound of* $[x_i]$*. Three cases may occur according to* $\underline{x_i}$*, as illustrated in Figure 4.3.*

In the cases 1 and 2 (i.e., intervals $[x_i]_1$ *and* $[x_i]_2$ *in the figure), and not in the case 3, there exists a point* $v_i \in [x_i]$ *which is a zero in* $[f^{x_i}_{max}]$ *(i.e., which belongs to the segment in bold). That is:*

$$\exists v_i \in [x_i] \ s.t. \ 0 \in [f^{x_i}_{max}](v_i) \tag{4.3}$$

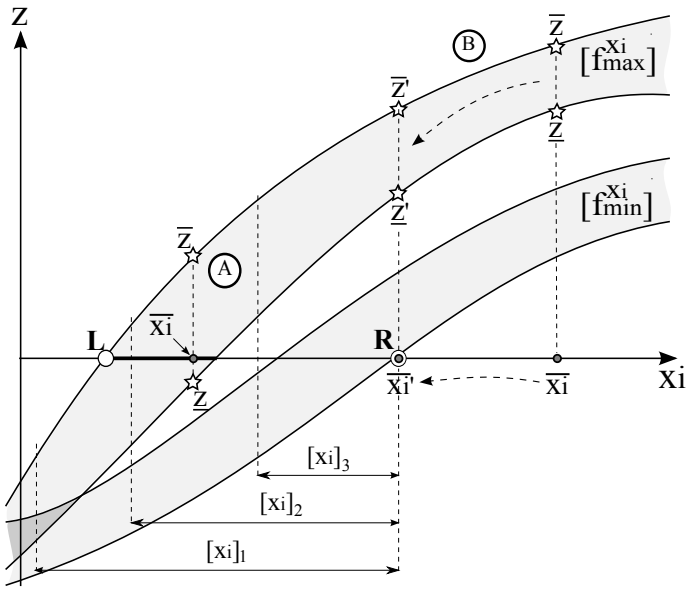

Figure 4.3: Two main cases in `Mohc-Revise` when `MaxRevise` is applied to f_{max}.

The relation (4.3) implies:

$$\forall j \neq i, \forall v_j \in [x_j] : \underline{[f^{x_j}_{min}](v_j)} \leq \underline{[f^{x_i}_{max}](v_i)} \leq 0$$

The rightmost inequality comes directly from (4.3). The leftmost inequality comes from the study of monotonicity of f: the values for X (excepting $x_i = v_i$) used by $[f^{x_i}_{max}](v_i)$ (see relation (4.1), page 77) maximize f while the values for X (excepting $x_j = v_j$) used by $[f^{x_j}_{min}](v_j)$ (see relation (4.2), page 77) minimize f. As $j \neq i$ $\underline{[f^{x_j}_{min}](v_j)} \leq \underline{[f^{x_i}_{max}](v_i)}$[1]. □

Thus, in the cases 1 and 2 of the figure, we know that $\underline{[f^{x_j}_{min}](v_j)} \leq 0$ for every $j \neq i$. This means that $[f^{x_j}_{min}]$ cannot bring any additional narrowing to $[x_j]$, the relation used to shave the interval being always true. In other terms, if x_j is increasing (resp. decreasing), then it is useless to call `RightNarrowFmin` (resp. `LeftNarrowFmin`) to contract $[x_j]$. The same reasoning is used symmetrically for predicting if `LeftNarrowFmax` (resp. `RightNarrowFmax`) will be useless.

4.4.4 Improvements

Lemmas 1, 2 and 3 lead to some improvements of our algorithm.

Improving the `MonotonicBoxNarrow` procedure

Lemmas 2 and 3 result in Algorithm 9, a more sophisticated version of the `MonotonicBoxNarrow` procedure described in Section 4.2.2.

The `MonotonicBoxNarrow` procedure contains two important improvements. First, if x_i is increasing then `LeftNarrowFmax` (resp. `RightNarrowFmin`) is called only if $\overline{[f_{min}]([B])} < 0$ (resp. $\underline{[f_{max}]([B])} > 0$), otherwise they are not useful (lines 2 and 9). This is due to the first case detailed in Section 4.4.2.

[1] We assume here that the interval gradient is computed using automatic differentiation. Otherwise, the punctual functions $\underline{[f^{x_j}_{min}]}$ and $\underline{[f^{x_j}_{min}]}$ may not have the same monotonicities as f.

Algorithm 9 `MonotonicBoxNarrow` (**in-out** $[B]$; **in** f_{max}, f_{min}, X, $[G]$, ϵ)

```
1:  for all variable x_i ∈ X do
2:    if overline([f_min]([B])) < 0 and applyFmax[i] then
3:      if [g_i] > 0 then
4:        LeftNarrowFmax([x_i], f_max^{x_i}, [g_i], ε)
5:      else
6:        RightNarrowFmax([x_i], f_max^{x_i}, [g_i], ε)
7:      end if
8:    end if
9:    if underline([f_max]([B])) > 0 and applyFmin[i] then
10:     if [g_i] > 0 then
11:       RightNarrowFmin([x_i], f_min^{x_i}, [g_i], ε)
12:     else
13:       LeftNarrowFmin([x_i], f_min^{x_i}, [g_i], ε)
14:     end if
15:   end if
16: end for
```

Second, the observations of Section 4.4.3 are also used to avoid unnecessary calls to the narrowing procedures. We add two new boolean arrays `applyFmin` and `applyFmax`. If x_i is increasing and the boolean `applyFmin[i]` (resp. `applyFmax[i]`) is set to *false* then the `LeftNarrowFmin` (resp. `RightNarrowFmax`) procedure is useless for narrowing $[x_i]$ (lines 2 and 9). For every monotonic variable $x_i \in X$, the booleans are initialized to *true* and are updated in the four procedures `LeftNarrowFmax`, `RightNarrowFmax`, `LeftNarrowFmin` and `RightNarrowFmin` in accordance with the last paragraph of Section 4.4.3.

This advanced feature is a slight generalization of the feature proposed in [Chabert and Jaulin, 2009b]. In their paper, $f_{min}^{x_j}$ and $f_{max}^{x_j}$ are punctual functions because there are no sets Y and W.

Lazy evaluations of f_{min} and f_{max}

`Mohc-Revise` is optimized in the aim of reusing as much as possible the different evaluations of f_{min} and f_{max}. In Algorithm 9, intervals $[z_{min}] = [f_{min}]([B])$ and $[z_{max}] = [f_{max}]([B])$ (lines 2 and 9) do not need to be recomputed at each iteration. Indeed, these intervals have been previously computed in the bottom-up evaluation phases of `MinMaxRevise` and can be transmitted to `MonotonicBoxNarrow`.

The value $\overline{z_{max}}$ can also be used to add a first and cheap call to a Newton iteration at beginning of `LeftNarrowFmax`. More precisely, we replace the third line in Algorithm 8 by: $[l] \leftarrow [x] \cap \left(\overline{x} - \frac{\overline{z_{max}}}{[g]}\right)$.

Finally, the existence test of `LeftNarrowFmax` makes it possible to reuse the value $[z_l] = [f_{max}^x](\underline{x})$. This allows us to add, just after the third line in Algorithm 8, a second cheap Newton iteration: $[l] \leftarrow [l] \cap \left(\underline{x} - \frac{\overline{z_l}}{[g]}\right)$.

The final version of the `LeftNarrowFmax` procedure is described in Appendix A.8.

The `LazyMohc` variant

`LazyMohc` is a simplified version of `Mohc` in which the `MonotonicBoxNarrow` procedure is replaced by the

`LazyMonotonicBoxNarrow` procedure (see the pseudocode in Appendix A.6). `LazyMonotonicBoxNarrow` only calls the first and cheap Newton iteration described above for every bound of x_i in X. In other words, the `LazyMohc` variant runs `MinMaxRevise`, two cheap Newton iterations per monotonic variable (the `LazyMonotonicBoxNarrow` procedure), but no dichotomic process.

4.5 Properties

A very interesting property concerning `Mohc` is that the `Mohc-Revise` procedure can compute an optimal box w.r.t. an individual constraint if certain conditions are fulfilled. These conditions thus identify a polynomial subclass (Proposition 10) of the hull-consistency problem, i.e., searching for the smallest box containing all the solutions of the constraint (hull-consistency is discussed in Section 3.2.3.2). Recall that the problem is difficult when there exist multiple occurrences of variables. (The corresponding propositions appear below and the proofs can be found in Appendix A.)

Proposition 10 *Let $c : f(X, Y, W) = 0$ be a constraint such that f is continuous and differentiable w.r.t. every variable in the box $[B] = [X] \times [Y] \times [W]$. Variables in X and W appear several times in f while variables $y_i \in Y$ appear once. For every $x_i \in X$, f is monotonic w.r.t. x_i in $[B]$.*

If W is empty and if for all $y_i \in Y$, $0 \notin \left[\frac{\partial f}{\partial y_i}\right]([B])$ (implying that y_i is monotonic), then:
One call to `Mohc-Revise` *computes the hull-consistency of the constraint c in $[B]$ (with a precision ϵ).*

The proof is based on the lemma 4 related to the box $[Y]$ contracted by `MinMaxRevise`, and on the lemma 5 related to the box $[X]$ contracted by `MonotonicBoxNarrow`.

Lemma 4 *With the same hypotheses as in Proposition 10, if W is empty and if for all $y_i \in Y$, $0 \notin \left[\frac{\partial f}{\partial y_i}\right]([B])$ (implying that y_i is monotonic), then:*

One call to `MinMaxRevise` *contracts optimally every $[y_i] \in [Y]$.*

Lemma 5 *With the same hypotheses as in Proposition 10:*

One call to `MonotonicBoxNarrow` *(following a call to* `MinMaxRevise`*) contracts optimally every $[x_i] \in [X]$.*

Proposition 11 is in a sense stronger than Proposition 10 because *no monotonicity hypothesis is required for the variables y_i occurring once* in the expression. However, a stronger and more expensive procedure is used instead of `HC4-Revise`. Replacing `HC4-Revise` with a so-called `TAC-revise` is a way to make a system hull-consistent when all the constraints contain only variables with single occurrence. The fact that a given function f, having only variables with single occurrence, is continuous ensures that the image produced by the natural extension $[f]$ is optimal. But the top-down narrowing phase manages in a sense inverse functions of f that are not necessarily continuous. `HC4-Revise` returns the hull of the different continuous subparts provided by the piecewise analysis performed at each node for the inverse functions. Instead, `TAC-revise` combinatorially combines the continuous subparts of different nodes for optimally narrowing the variables, thus achieving hull-consistency. Details about `TAC-revise` can be found in [Chabert et al., 2005] and in Section 3.2.3.4.

Proposition 11 *Let $c : f(X, Y, W) = 0$ be a constraint such that f is continuous and differentiable w.r.t. every variable in the box $[B] = [X] \times [Y] \times [W]$. Variables in X and W appear several times in f while*

variables $y_i \in Y$ appear once. For every $x_i \in X$, f is monotonic w.r.t. x_i. Let us call `Mohc-Revise'` *a variant of* `Mohc-Revise` *in which* `HC4-Revise` *is replaced by* `TAC-revise` *in* `MinMaxRevise`.

If W is empty, then:
One call to `Mohc-Revise'` *computes the hull-consistency of the constraint c in $[B]$ (with a precision ϵ).*

This proposition is significant because in practice, after only a few bisections in the search tree, `HC4-Revise` generally computes a box as sharp as `TAC-revise` does, except in pathological cases. Thus, `Mohc-Revise` (calling the standard `HC4-Revise` procedure) *often* computes hull-consistency when all the variables appear once or are monotonic.

Finally, Proposition 12 details the time complexity of `Mohc-Revise`.

Proposition 12 *Let c be a constraint. Let n be its number of variables, e be the number of unary and binary operators in the constraint (i.e., at least equal to the total number of occurrences of variables), and k be the maximum number of occurrences of a variable (thus, $n\,k \leq e$). Let ϵ be the precision expressed as a ratio of interval diameter.*

With no call to `OccurrenceGrouping`, `Mohc-Revise` *is time $O(n\,e\,log(\frac{1}{\epsilon}))$.*
`LazyMohc-Revise` *is time $O(e+n) = O(e)$.*

With `OccurrenceGrouping`, `Mohc-Revise` *is time $O(n\,(e\,log_2(\frac{1}{\epsilon}) + k\,log_2(k))) = O(n\,e\,log_2(\frac{k}{\epsilon}))$.*
`LazyMohc-Revise` *is time $O(e+n+n\,k\,log_2(k)) = O(e\,log_2(k))$.*

4.6 Experiments

We have implemented `Mohc` and `LazyMohc` with the interval-based `C++` library `Ibex` [Chabert, 2009; Chabert and Jaulin, 2009a]. `Mohc` has been tested on 17 benchmarks with a finite number of zero-dimensional solutions issued from COPRIN's web page [Merlet, 2009]. We have tested `Mohc` in two different ways: as a main contractor between two bisections or as a subcontractor embedded in the `3BCID` contractor (described in Section 3.2.4.2).

We have selected all the NCSPs with variables with more than two occurrences in a same constraint which have been found in the first two sections (polynomial and non polynomial systems) of the web page. We have added `Brent`, `Butcher`, `Direct Kinematics` and `Virasoro` from the section describing the *difficult problems*. All the competitors are also available in `Ibex`, thus making the comparison fair.

All the experiments have been performed on an `Intel 6600 2.4 GHz`, and a timeout of at least one hour has been chosen for each benchmark.

4.6.1 `Mohc` as a subcontractor of `3BCID`

4.6.1.1 Tuning the user-defined parameters

The first experiments allow us to get an idea of relevant values for τ_{mohc} (see Section 4.3.1) and ϵ. The curves of Figure 4.4-left show how the ratio $\frac{Time(Mohc)}{Time(LazyMohc)}$ evolves when ϵ decreases (i.e., when the reached precision in `MonotonicBoxNarrow` increases). It appears that tuning ϵ has no significant impact on performance. For most of the NCSPs, the best value falls between $\frac{1}{32}$ and $\frac{1}{8}$, and the curves are rather flat. We have thus fixed ϵ to 10% (at least when `Mohc` is a subcontractor of `3BCID`).

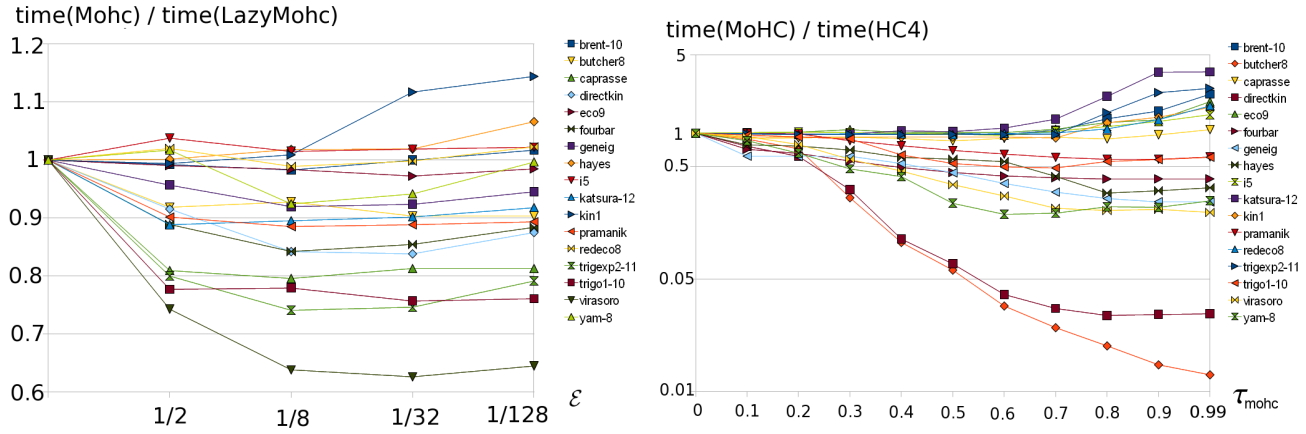

Figure 4.4: Tuning the user-defined parameters. **Left:** Tuning ϵ. **Right:** Tuning τ_{mohc}.

The curves of Figure 4.4-right show how the ratio $\frac{Time(Mohc)}{Time(HC4)}$ evolves when τ_{mohc} increases, i.e., when the monotonic-based procedures are called more often. For all the NCSPs, the best value of τ_{mohc} falls between 0.6[1] and 0.99. Thus, the experiments that follow perform two trials with $\tau_{mohc} = 0.7$ and $\tau_{mohc} = 0.99$ (for the most favorable NCSPs).

4.6.1.2 Experimental protocol

Our `Mohc`-based solving strategy uses a round-robin variable selection. Between two branching points, three procedures are called in sequence. First, a monotonicity test `MonoTest` just checks whether every monotonicity-based function evaluation contains zero. Its specificity is that the test does not apply to the initial functions f in the NCSP, but to the functions f^{og} produced by `OccurrenceGrouping`. Second, the contractor `3BCID(Mohc)` is called. Third, an interval Newton is run if the current box has a diameter 10 or less. All the parameters in `3BCID`, `HC4`, `Box` and `Mohc` (except τ_{mohc}) have been fixed to default values. The precision ratio in `3B` and `Box` is 10% ; the number of additional slices handled by the `CID` part is 1; a constraint is pushed into the propagation queue if the interval of one of its variables is reduced more than $\tau_{propag} = 10\%$.

4.6.1.3 Results

Table 4.1 compares the CPU time and number of choice points obtained by `3BCID(Mohc)` with those obtained by competitors: `3BCID(HC4)` and `3BCID(Box)`.

The first column includes the name of the benchmark; the bottom of the cell contains the corresponding number of equations and the number of solutions. The other columns report the results obtained by different algorithms. Every cell shows the CPU time in second (above) and the number of choice points (below). The contraction algorithms are `3BCID(HC4)` (column `HC4`), `3BCID(Box)` (column `Box`), `MonoTest` followed by `3BCID(HC4)` (column `MonoTest`). `MonoTest` is also used before the contractor shown in the

[1]Observe that, even on NCSPs that do not benefit from our monotonicity-based procedures, setting τ_{mohc} to 0.6 only slightly decreases the performance (only 11% in the worst case).

NCSP	HC4	Box	MonoTest	Lazy(0.7)	Lazy(0.99)	Mohc(0.7)		Mohc(0.99)		Mohc(1)
							Gain		Gain	
Butcher	282528	25867	281664	5220	1985	5431	52.0	**1722**	163	1915
8 3	1.8e+8	1.7e+6	1.8e+8	2.2e+6	346063	2.2e6	82	288773	623	273049
Direct kin.	17515	>28800	17507	480	431	429	40.8	**356**	49.1	401
11 2	1.4e+6		1.4e+6	11931	8811	8859	157	5503	253	5487
Virasoro	7158	>28800	7173	1537	1413	1052	6.82	**897**	8.00	**899**
8 224	2.6e+6		2.6e+6	135833	102997	71253	36.0	38389	66.8	38391
Geneig	590	>7200	390	116	95.2	108	3.62	**81.1**	4.81	97.0
6 10	205263		161211	17059	9083	13909	11.6	6061	26.7	6233
Yamam.1	11.2	15.3	11.7	**2.26**	2.91	**2.20**	5.30	2.87	4.06	3.59
8 7	3013	183	3017	357	345	345	8.74	295	10.2	267
Fourbar	13121	11011	1069	429	419	**366**	2.92	**372**	2.87	377
4 3	8.5e+6	732429	965343	79697	67397	58571	16.5	45561	21.2	45475
Hayes	39.2	282	41.6	17.2	**13.6**	17.0	2.44	**13.8**	3.02	16.0
8 1	17649	7247	17763	4447	1707	4375	4.06	1679	10.6	1725
Trigo1	160	773	151	74.0	92.3	**57.7**	2.61	73.2	2.06	73.3
10 9	2791	1005	2565	641	603	459	5.59	443	5.79	443
Pramanik	100	278	35.9	21.9	22.1	**20.8**	1.72	**21.3**	1.69	26.1
3 2	124661	23017	69259	13817	9649	12691	5.46	8429	8.22	8219
Caprasse	**2.56**	32.2	2.73	**2.51**	2.94	2.64	1.03	4.35	0.63	4.60
4 18	1305	719	1309	951	499	867	1.51	383	3.42	383
Redeco8	**5.80**	69.8	6.28	6.32	11.0	6.10	1.03	10.7	0.59	16.5
8 8	2295	1913	2441	2231	1741	2211	1.10	1489	1.64	1331
Kin1	**1.73**	68.7	1.96	**1.78**	3.33	**1.79**	1.09	3.43	0.57	5.47
6 16	85	65	87	83	83	83	1.05	83	1.05	71
Trigexp2	**82.7**	>3600	86.9	88.2	228	87.0	1.00	164	0.53	163
11 0	14283		14299	14303	12567	14299	1.00	7291	1.96	6597
I5	**54.6**	>3600	55.9	58.4	81.7	57.5	0.97	84.1	0.66	115
10 30	10595		10621	9809	8849	9773	1.09	8693	1.22	7561
Eco9	**13.9**	102	**13.9**	15.1	26.5	**14.0**	0.99	26.6	0.52	38.7
9 16	6193	4991	6193	6047	4707	6025	1.03	4309	1.44	4035
Brent	**18.1**	311	18.9	20.0	42.1	19.9	0.95	41.4	0.46	46.5
10 1008	3953	2137	3923	3807	3341	3805	1.03	3189	1.23	2377
Katsura	**73.2**	2265	77.8	104	274	103	0.75	251	0.31	260
12 7	4265	3557	4251	3671	3373	3573	1.19	3471	1.22	347

Table 4.1: Results obtained by Mohc as a subcontractor of 3BCID.

four last columns: 3BCID(LazyMohc) with $\tau_{mohc} = 0.7$ (column Lazy$_{(0.7)}$), the same with $\tau_{mohc} = 0.99$ (column Lazy$_{(0.99)}$), 3BCID(Mohc) with $\tau_{mohc} = 0.7$ and $\epsilon = 10\%$ (column Mohc$_{(0.7)}$-left), the same with $\tau_{mohc} = 0.99$ (column Mohc$_{(0.99)}$-left). All the algorithms are followed by a call to interval Newton before the next bisection. The columns Mohc$_{(0.7)}$-right and Mohc$_{(0.99)}$-right yield the gain obtained by Mohc (with $\tau_{mohc} = 0.7$ and $\tau_{mohc} = 0.99$ resp.) w.r.t. MonoTest followed by 3BCID(HC4), i.e., $\frac{Time(\texttt{MonoTest})}{Time(\texttt{3BCID(Mohc)})}$ (above) and $\frac{Choice\_Points(\texttt{MonoTest})}{Choice\_Points(\texttt{3BCID(Mohc)})}$ (below). Observe that MonoTest followed by 3BCID(HC4) is the best competitor strategy, especially in the benchmarks Fourbar, Geneig and Pramanik with gains w.r.t. 3BCID(HC4) of 12, 1.5 and 3.7 respectively. The worst gain is 0.9 and has been observed for Kin1.

The table highlights the very good results obtained by 3BCID(Mohc), both in terms of filtering power (low number of choice points) and CPU time. As expected, the bad results obtained by Box highlight that Box is not relevant when several variables in a same constraint have multiple occurrences. For the NCSPs Yamamura1, Brent and Kin1, Box shows a slightly better contraction power than Mohc, but it does not pay off in terms of performance. This underlines that it is better to perform a box narrowing effort less often, when monotonicity has been detected for a given variable.

The comparison with MonoTest followed by 3BCID(HC4) is more interesting. Mohc$_{(0.7)}$ and MonoTest obtain similar results on 8 of the 17 benchmarks. Note that the loss in performance of Mohc$_{(0.7)}$ is negligible. The gain is clearly inferior than 1 only for Katsura. On 7 NCSPs, Mohc$_{(0.7)}$ and Mohc$_{(0.99)}$ show a gain comprised between 1.7 and 8. On Butcher and Direct Kin., a very good gain in CPU time of resp. 163 and 49 (Mohc$_{(0.99)}$) or 52 and 42 (Mohc$_{(0.7)}$) is observed.

Although the difference is not so significant, 3BCID(Mohc) generally provides better results than 3BCID(LazyMohc), in particular when $\tau_{mohc} = 0.99$.

The last column (Mohc$_{(1)}$) shows that setting τ_{mohc} to 1 (i.e., ignoring completely the parameter) is slightly but systematically worse than setting it to 0.99.

4.6.1.4 Profiling

In this section we report the results of different profiling tests. They aim at evaluating the impact on performance of the different features of Mohc. All the results have been produced using the 3BCID(Mohc) strategy and $\tau_{mohc} = 0.99$. We call MonotonicProcedures the virtual procedure grouping the set of the monotonicity-based procedures inside Mohc-Revise: GradientCalculation, OccurrenceGrouping, ExtractMonotonicVars, MinMaxRevise and MonotonicBoxNarrow.

CPU time distribution

Table 4.2 reports the CPU time distribution of the different procedures inside Mohc-Revise.

The first column includes the name of the benchmark. The other columns report the ratio ($\frac{total\_time(P)}{total\_time(\texttt{Mohc})}$) between the time taken by the sum of calls to a given procedure P (from columns 2 to 7, P is: HC4-Revise, GradientCalculation, OccurrenceGrouping + ExtractMonotonicVars, MinMaxRevise, MonotonicBoxNarrow and MonotonicProcedures) and the time taken by the sum of calls to Mohc-Revise during the resolution of each benchmark. The last row corresponds to the average of each column.

Consider t the time spent by Mohc in the solving of one benchmark[1]. It is interesting to observe that the

[1]For every benchmark, Mohc takes between 80% and 97% of the total solving time. The rest of time is shared by interval Newton and the monotonicity-based existence test.

		MonotonicProcedures				
NCSP	HC4-Revise	Grad.Calc.	OG	MinMaxR.	Mono.BoxNarrow	Mono.Proc.
Brent	**0.31**	0.21	0.04	0.30	0.14	**0.69**
Caprasse	**0.28**	0.19	0.08	0.29	0.14	**0.72**
Eco9	**0.29**	0.15	0.06	0.43	0.07	**0.71**
Geneig	**0.25**	0.19	0.10	0.33	0.13	**0.75**
Hayes	**0.27**	0.24	0.04	0.36	0.08	**0.73**
I5	**0.53**	0.11	0.04	0.30	0.02	**0.47**
Katsura	**0.21**	0.21	0.04	0.38	0.16	**0.79**
Kin1	**0.36**	0.27	0.00	0.32	0.05	**0.64**
Pramanik	**0.28**	0.20	0.14	0.36	0.03	**0.72**
Redeco8	**0.29**	0.14	0.09	0.35	0.12	**0.71**
Trigexp2	**0.23**	0.26	0.05	0.31	0.15	**0.77**
Trigo1	**0.23**	0.24	0.00	0.46	0.07	**0.77**
Yamamamura1	**0.31**	0.20	0.04	0.38	0.07	**0.69**
Fourbar	**0.26**	0.22	0.12	0.32	0.07	**0.74**
Direct Kin.	**0.25**	0.29	0.02	0.33	0.11	**0.75**
Butcher	**0.25**	0.18	0.06	0.49	0.01	**0.75**
Virasoro	**0.19**	0.21	0.10	0.32	0.18	**0.81**
AVERAGE	**0.28**	0.21	0.06	0.35	0.09	**0.72**

Table 4.2: Time distribution of Mohc-Revise procedures.

HC4-Revise procedure takes a significant part of the time (on average 28% of t). If the monotonicity-based procedures *are completely useless* for narrowing domains further, then the loss in performance w.r.t. 3BCID(HC4) is limited to 0.28 (on average). In Table 4.1 we can see the worst results in Katsura where the gain is 0.32.

The procedure spending most time in the resolution is MinMaxRevise (35% of t). OccurrenceGrouping takes only 6% of t. Surprisingly, MonotonicBoxNarrow takes 10% of t, a very little time if we consider that it can reach the hull-consistency (see Section 7.3.5).

Application frequency of the monotonicity-based procedures

The curves of Figure 4.5 show how the application frequency of MonotonicProcedures ($\frac{number\_of\_calls(\texttt{MonotonicProcedures})}{number\_of\_calls(\texttt{Mohc-Revise})}$) evolves when τ_{mohc} increases. Low values for τ_{mohc} imply low application frequencies of MonotonicProcedures, just like high values of τ_{mohc} imply high application frequencies of MonotonicProcedures. In particular when $\tau_{mohc} = 1$, the monotonic-based procedures are applied in each call to Mohc-Revise.

The figure should also be understood in a different way: the application frequency for a benchmark with a given τ_{mohc} (e.g., 0.5) corresponds, *more or less*, to the frequency in which ρ_{mohc} is less than τ_{mohc}, i.e., the diameter of the evaluation by monotonicity is 50% or less than the diameter of the natural evaluation (see Section 4.3.1 for an explanation of the parameter τ_{mohc}).

Observe that when τ_{mohc} is 0.7 the application frequency of MonotonicProcedures for Katsura, I5, Brent, Redeco8, Kin1, Eco9 and Trigexp2 is less than 25%. All these benchmarks show a decrease in performance when the frequency increases (see columns Mohc$_{(0.99)}$ and Mohc$_{(1)}$ of Table 4.1). On the

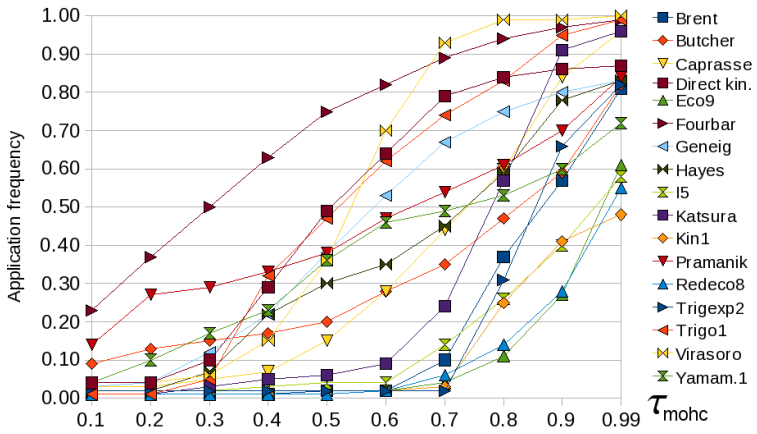

Figure 4.5: Application frequency of the monotonicity-based procedures in function of τ_{mohc}.

contrary, the benchmarks having higher application frequencies of `MonotonicProcedures` show good results when τ_{mohc} is 0.7 or more (excepting `Caprasse`).

Thus, it seems that the array ρ_{mohc} offers, in general, a good prediction of the behavior of `MonotonicProcedures`. This leads to the opportunity of an adaptive tuning of τ_{mohc} (between 0.6 and 0.99).

Remark in Figure 4.5, even when τ_{mohc} is close to 1, that there exist application frequencies with values less than 90%. Setting τ_{mohc} to 1 implies an application frequency of 100% resulting in a decrease in performance due to the big additional effort and probably poor additional contraction (compare columns $\texttt{Mohc}_{(0.99)}$ and $\texttt{Mohc}_{(1)}$ in Table 4.1). This explains why removing τ_{mohc} (or defining $\tau_{mohc} = 1$) reports, in general, worse results than setting τ_{mohc} to 0.99.

Time of the monotonic-based procedures w.r.t. `HC4-Revise`

Table 4.3 reports the ratio $(\frac{time(P)}{time(\texttt{HC4-Revise})})$ between the *average* time taken by *one call* to a given procedure P (from columns 2 to 5, P is: `GradientCalculation`, `OccurrenceGrouping` + `ExtractMonotonicVars`, `MinMaxRevise` and `MonotonicBoxNarrow`) and *one call* to `HC4-Revise` (observe that this is not equivalent to the results reported in Table 4.2 because, in general, each procedure is called a different number of times). The last column reports the ratio between one call to all the procedures inside `Mohc-Revise` (including `HC4-Revise`) and *one call* to `HC4-Revise`. The column $(\#X)$, beside `Mono.BoxNarrow`, reports the average number of variables treated by one call to `MonotonicBoxNarrow` (i.e., the average number of variables in X).

As expected, the time taken by `GradientCalculation` is close to the time taken by `HC4-Revise`. Recall that the gradient calculation is performed by the backward AD method (see Section 2.2.4.2). It traverses the expression tree twice, like `HC4-Revise`, but performing different operations.

Remark that the `MinMaxRevise` procedure takes about twice the time of `HC4-Revise`. It seems logical because `MinMaxRevise` uses `HC4-Revise` for contracting 2 inequalities: $f_{min}(X, Y, W) \leq 0$ and $f_{max}(X, Y, W) \geq 0$. The interesting thing is that f_{min} and f_{max} are auxiliary functions that contain more atomic operations than f (recall that they are obtained by the `OccurrenceGrouping` procedure). Thus, the experiments show that these new operations do not decrease, in a significant way, the perfor-

NCSP	Grad.Calc.	OG	MinMaxR.	Mono.BoxNarrow	($\#X$)	Mohc
Brent	0.99	0.21	1.42	0.66	(2.92)	4.27
Caprasse	0.76	0.33	1.15	0.59	(1.73)	3.84
Eco9	1.19	0.52	3.41	0.55	(1.42)	6.68
Geneig	1.07	0.54	1.83	0.73	(4.25)	5.17
Hayes	1.27	0.22	1.92	0.43	(3.34)	4.84
I5	0.49	0.2	1.4	0.09	(0.31)	3.18
katsura	1.1	0.2	1.97	0.84	(4.82)	5.11
Kin1	2.37	0.04	2.74	0.41	(2.92)	6.56
Pramanik	0.98	0.67	1.76	0.2	(0.33)	4.61
Redeco8	1.34	0.85	3.28	1.19	(1.98)	7.67
Trigexp2	1.6	0.3	1.87	0.95	(2.00)	5.72
Trigo1	1.02	0.02	1.99	0.31	(0.78)	4.34
Yamam.1	1.08	0.24	2.11	0.39	(0.60)	4.82
Fourbar	0.87	0.5	1.29	0.37	(1.70)	4.02
Direct kin.	1.5	0.11	1.72	0.59	(3.59)	4.93
Butcher	0.95	0.34	2.64	0.08	(0.37)	5.01
Virasoro	1.07	0.53	1.66	0.97	(4.71)	5.23
AVERAGE	**1.16**	**0.34**	**2.01**	**0.55**	(**2.22**)	**5.06**

Table 4.3: Time ratios of monotonic procedures compared to HC4-Revise.

mance of MinMaxRevise.

Observe the interesting results of MonotonicBoxNarrow. In most of the cases, the procedure takes in general less time than HC4-Revise for treating several variables (recall MonotonicBoxNarrow applies a narrowing procedure to each bound of the monotonic variables, each narrowing procedure consisting in several evaluations and Newton iterations). This nice behavior is mainly due to the improvements proposed in Section 4.4.4 that avoid a lot of calls to the narrowing procedures while maintaining the same contraction power.

4.6.2 Mohc as the main contractor

When Mohc is the only contractor (together with Newton), the time of the preprocessing procedure calculating $\rho_{mohc}[f]$ for each constraint f (see Section 4.3.1) is not negligible. Furthermore, the preprocessing calls the same monotonicity-based procedures as Mohc-Revise except MonotonicBoxNarrow.

This observation allows us to optimize Mohc when it is applied only once between bisections (and not inside 3BCID). The improvement consists in using Algorithm 10, instead of Mohc-Revise (Algorithm 6), the first time f is revised after a bisection. Mohc-Revise$_\rho$ computes $\rho_{mohc}[f]$ while it performs the revise of the function (line 8). The other times f is revised in the same propagation process, the standard Mohc-Revise is used.

If $W \neq \emptyset$, then Mohc-Revise$_\rho$ performs all the monotonicity-based procedures required for computing $\rho_{mohc}[f]$. $\rho_{mohc}[f]$ is then computed before calling MonotonicBoxNarrow. If $\rho_{mohc}[f] \leq \tau_{mohc}[f]$ then MonotonicBoxNarrow is executed, otherwise the procedure terminates.

Algorithm 10 `Mohc-Revise`$_\rho$ (**in-out** $[B]$, ρ_{mohc}; **in** f, Y, W, τ_{mohc}, ϵ)

1: `HC4-Revise`$(f(Y,W)=0, Y, W, [B])$
2: **if** $W \neq \emptyset$ **then**
3: $([G],[G_o]) \leftarrow$ `GradientCalculation`$(f, W, [B])$
4: $(f^{og}, W) \leftarrow$ `OccurrenceGrouping`$(f, W, [B], [G_o])$
5: $(f_{max}, f_{min}, X, W) \leftarrow$ `ExtractMonotonicVars`$(f^{og}, W, [B], [G])$
6: `MinMaxRevise`$([B], f_{max}, f_{min}, Y, W)$
7: `LazyMonotonicBoxNarrow`$([B], f_{max}, f_{min}, X, [G], \epsilon)$
8: $\rho_{mohc}[f] \leftarrow \dfrac{Diam([[\underline{f_{min}}]([B]), \overline{[f_{max}]}([B])])}{Diam([f]([B]))}$
9: **if** $\rho_{mohc}[f] \leq \tau_{mohc}$ **then**
10: `MonotonicBoxNarrow`$([B], f_{max}, f_{min}, X, [G], \epsilon)$
11: **end if**
12: **end if**

4.6.2.1 Tuning the user-defined parameters

The curves of Figure 4.6-left show how the ratio $\frac{Time(Mohc)}{Time(LazyMohc)}$ evolves when ϵ decreases (i.e., the reached precision in `MonotonicBoxNarrow` increases). Similarly to Figure 4.4-left, finely tuning ϵ has no significant impact on performance. For most of the NCSPs, the best value falls near $\frac{1}{32}$. For these experiments, we have thus fixed ϵ to 3%.

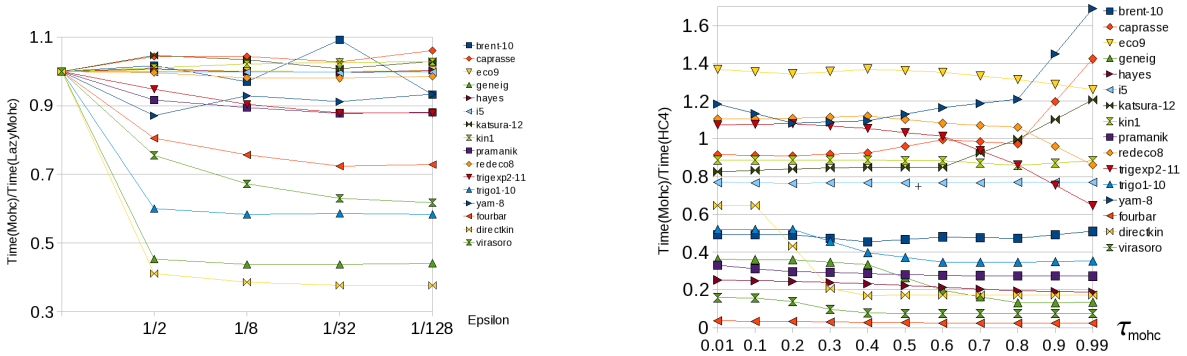

Figure 4.6: Tuning the user-defined parameters. **Left:** Tuning ϵ. **Right:** Tuning τ_{mohc}.

The curves of Figure 4.6-right show how the ratio $\frac{Time(Mohc)}{Time(HC4)}$ evolves when τ_{mohc} increases. Note that increasing the value of τ_{mohc} slightly improves the performance of `Mohc` in most of the benchmarks. However, some few benchmarks show a decreasing in the performance with high values of τ_{mohc} (e.g., `Yamamura1`, `Caprasse`, `Katsura`). The experiments that follow perform thus two trials with the same parameter values used by `3BCID(Mohc)`, i.e., $\tau_{mohc} = 0.7$ and $\tau_{mohc} = 0.99$.

4.6.2.2 Experimental protocol

Our `Mohc`-based solving strategy uses a round-robin variable selection. Between two branching points, two procedures are called in sequence. First, the contractor `Mohc` (the variant including the preprocessing

procedure) is called. Then, an interval Newton is executed.

NCSP	HC4	MT+HC4	Box	MT+Box	Lazy(0.7)	Lazy(0.99)	Mohc(0.7)		Mohc(0.99)	
								Gain		Gain
Virasoro	>14400	>14400	>14400	>14400	1879	1845	1180	12	**1090**	13
8 224					1.7e+6	1.7e+6	805047		715407	
Direct kin.	>14400	>14400	>14400	>14400	6800	6784	2560	5.6	**2481**	5.8
11 2					2.2e+6	2.2e+6	777281		730995	
Hayes	147	163	311	323	31.1	29.4	30.9	4.7	**27.6**	5.3
8 1	560923	541817	214247	214253	74613	59429	73317	2.9	49059	4.4
Geneig	3244	1966	7726	3721	1057	1010	463	4.2	**436**	4.5
6 10	6.4e+6	4.1e+6	2.2e+6	1.3e+6	1.9e+6	1.7e+6	799439	1.6	655611	2.0
Trigo1	86.7	93	324	332	51.1	56.1	**30.0**	2.9	**30.6**	2.8
10 9	5725	5725	6241	6241	2481	2411	1759	3.3	1673	3.4
Fourbar	>14400	864	>14400	2442	497	498	361	2.4	**359**	2.4
4 3		1.6e+6		1.2e+6	728533	725045	437959	2.7	430847	2.7
Trigexp2	1592	1610	>14400	>14400	1718	1830	1507	1.1	**1027**	1.5
11 0	1.7e+6	1.7e+6			1.6e+6	1.6e+6	1.4e+6	1.2	935227	1.8
I5	9240	9310	>14400	>14400	7203	**7052**	7107	1.3	7129	1.3
10 30	2.5e+7	2.4e+7			1.6e+7	1.5e+7	1.6e+7	1.5	1.5e+7	1.6
Redeco8	3408	3769	9676	9906	3601	3561	3529	1.0	**2936**	1.2
	1.0e+7	1.0e+7	8.0e+6	8.0e+6	7.0e+6	6,2e+6	6.8e+6	1.2	4.7e+6	1.7
Kin1	6.40	6.91	26.6	26.9	**5.64**	6.15	5.76	1.1	**5.65**	1.1
6 16	1309	1303	689	689	1021	963	1017	0.7	931	0.7
Pramanik	91.8	26.9	262	91.9	28.9	28.0	**25.3**	1.1	**25.0**	1.1
3 2	487271	103827	178887	81865	83921	83763	69809	1.2	69637	1.2
Katsura	119	182	2239	2286	**106**	150	**106**	1.1	143	0.8
12 7	271955	271493	251727	251727	102555	95583	98779	2.5	94249	2.7
Caprasse	1.89	2.04	12.0	11.5	**1.82**	4.54	**1.87**	1.0	2.69	0.7
4 18	8609	7671	6229	5957	5069	4751	4577	1.3	3741	1.6
Eco9	**35.1**	39.9	90.2	94.1	47.1	45.3	46.8	0.7	44.2	0.8
9 16	115957	115445	110857	110423	98211	90727	97961	1.1	84457	1.3
Brent	456	497	**150**	**151**	224	369	244	0.6	232	0.6
10 1008	1.9e+6	1.9e+6	23855	23855	677309	1.1e+6	752533	0.0	645337	0.0
Yamam.1	16.0	32.4	**12.3**	**12.6**	20.9	25.1	19.2	0.6	27.0	0.5
8 7	29645	29513	3925	3925	26739	26165	24767	0.2	29973	0.1

Table 4.4: Results obtained by Mohc.

All the parameters in HC4, Box and Mohc have been fixed to default values. The precision ratio in Box is 10%; a constraint is pushed into the propagation queue if the interval of one of its variables is reduced more than $\tau_{propag} = 1\%$.

The propagation queue of HC4, Box and Mohc and the list of constraints of MonoTest are initialized in an incremental way, i.e., using only the constraints related to the last bisected variable.

4.6.2.3 Results

Table 4.4 compares the CPU time and number of choice points obtained by `Mohc` with those obtained by competitors: `HC4` and `Box`.

The first column includes the name of the benchmark; the bottom of the cell contains the corresponding number of equations and the number of solutions. The other columns report the results obtained by different algorithms. Every cell shows the CPU time in second (above) and the number of choice points (below). The contraction algorithms are `HC4`, `Box`, `MonoTest` followed by `HC4` (column `MT+HC4`), `MonoTest` followed by `Box` (column `MT+Box`), `LazyMohc` with $\rho_{mohc} = 0.7$ (column $\texttt{Lazy}_{(0.7)}$), `LazyMohc` with $\rho_{mohc} = 0.99$ (column $\texttt{Lazy}_{(0.99)}$), `Mohc` with $\tau_{mohc} = 0.7$ and $\epsilon = 3\%$ (column $\texttt{Mohc}_{(0.7)}$-left), the same with $\tau_{mohc} = 0.99$ (column $\texttt{Mohc}_{(0.99)}$-left). All the algorithms are followed by a call to interval Newton before the next bisection. The columns $\texttt{Mohc}_{(0.7)}$-right and $\texttt{Mohc}_{(0.99)}$-right yields the gain obtained by `Mohc` (with $\tau_{mohc} = 0.7$ and $\tau_{mohc} = 0.99$ resp.) w.r.t. the best time/choice_points of the competitors `HC4`, `MT+HC4`, `Box` and `MT+Box`, i.e., $\frac{Min(Time(competitors))}{Min(Time(\texttt{Mohc}))}$ (above) and $\frac{Min(Choice\_Points(competitors))}{Min(Choice\_Points(\texttt{Mohc}))}$ (below).

The table reports good results obtained by `Mohc` as the main contractor, both in terms of filtering power (low number of choice points) and CPU time. In 6 of the 16 benchmarks (`Butcher` is not included because it takes more than 4 hours for all the strategies), the gains of $\texttt{Mohc}_{(0.7)}$ and $\texttt{Mohc}_{(0.99)}$ w.r.t. the competitors is more than 2. In only 3 benchmarks for $\texttt{Mohc}_{(0.7)}$, and 5 for $\texttt{Mohc}_{(0.99)}$, the performance gets worse (at most 50% in `Yamam.1`).

4.7 Advanced `MinMaxRevise'` procedure

In this section, we describe an improvement of the `MinMaxRevise` procedure (see Section 4.2.1). The improvement allows us to contract the intervals in $[X]$ as well (recall that `MinMaxRevise` only contracts intervals in $[Y]$ and $[W]$). This advanced `MinMaxRevise'` procedure is a work ongoing, so that it has not been already implemented nor tested.

In the following we will assume a function f monotonic *increasing* w.r.t. a variable x. The extension of the properties and the procedure to monotonic decreasing variables is straightforward.

4.7.1 A motivating example

Consider the constraint $4x - y^2x + x^2 - 40 = 0$ with the related intervals $[x] = [1, 5]$ and $[y] = [0, 2]$. The interval partial derivative w.r.t. x in the current box is $4 - [y]^2 + 2[x] = [2, 14]$. x is a monotonic increasing variable then we can use `MinMaxRevise` for contracting the interval $[y]$. `MinMaxRevise` uses punctual values instead of the interval $[x]$ ($\underline{x} = 1$ in the `MinRevise` procedure and $\overline{x} = 5$ in the `MaxRevise` procedure). For this reason, it cannot contract $[x]$.

The new `MinMaxRevise'` procedure uses for each occurrence x' of x a different constraint/function (**implicit constraint/function**). For instance, the implicit function related to the third occurrence of x is:

$$f_{x'}(x, x') = 4x - y^2x + x'^2 - 40 \tag{4.4}$$

Analogously to the `MinMaxRevise` procedure, `MinMaxRevise'` uses two procedures for contracting the box: `MinRevise'` and `MaxRevise'`. Thus, for contracting x', the `MaxRevise'` procedure works with the constraint

$$4\overline{x} - y^2\overline{x} + x'^2 - 40 \geq 0$$

Observe that, as f is monotonic increasing w.r.t. x, the variable is replaced by the right bound of its related interval (like in the MaxRevise procedure).

x' appears once in the implicit constraint, then the interval $[x']$ can be contracted using the monotonicity of x (if $f_{x'}$ is *monotonic increasing* w.r.t. x). The following proposition describes a cheap method to check if an implicit function $f_{x'}$ is monotonic increasing w.r.t. x.

Proposition 13 *Consider a function f. $f_{x'}$ is an implicit function of f related to an occurrence x' of a variable x. Assume that we compute the interval gradient using the automatic differentiation method (described in Section 2.2.4.2).*

If f is monotonic increasing w.r.t. x and

$$\underline{g_x} - \underline{g_{x'}} \geq 0 \tag{4.5}$$

where $[g_x] = \left[\frac{\partial f}{\partial x}\right]([B])$ and $[g_{x'}] = \left[\frac{\partial f}{\partial x'}\right]([B])$.

Then $f_{x'}$ is also monotonic increasing w.r.t. x.

Proof 8 *Consider x_O the set of occurrences of x in f excepting x'. As the gradient is computed using AD[1], then the interval partial derivative of the implicit function w.r.t. x is $[g_x^{f_{x'}}] = \left[\frac{\partial f_{x'}}{\partial x}\right]([B]) = \sum_{x_o \in x_O} \left[\frac{\partial f}{\partial x_o}\right]([B])$. Also using AD we obtain $[g_x] = [g_x^{f_{x'}}] + [g_{x'}]$ (i.e., $\underline{g_x} - \underline{g_{x'}} = \underline{g_x^{f_{x'}}}$). Finally,*

$$\underline{g_x} - \underline{g_{x'}} \geq 0 \Rightarrow \underline{g_x^{f_{x'}}} \geq 0 \Rightarrow \left[\frac{\partial f_{x'}}{\partial x}\right]([B]) \geq 0 \quad \square$$

In the example, $[g_x] = [2, 14]$ and $[g_{x'}] = 2 \times [y] = [2, 10]$, then

$$\underline{g_x} - \underline{g_{x'}} = 2 - 2 \geq 0$$

As $f_{x'}$ is monotonic increasing w.r.t. x', we can project the constraint $4\overline{x} - y^2\overline{x} + x'^2 - 40 \geq 0$ ($4\overline{x} - [y]^2\overline{x} + [x']^2 - 40 \geq 0 = [0, +\infty]$) over the occurrence x':

$$[x'] \leftarrow [x'] \cap \sqrt{[0, +\infty] - 4 \times \overline{x} + [y]^2\overline{x}} = [1, 5] \cap [4.47, +\infty] = [4.47, 5] \tag{4.6}$$

4.7.2 Evaluations and projections in MinMaxRevise'

As mentioned above, MinMaxRevise' executes the two procedures MinRevise' and MaxRevise' for intervals related to variables in X and Y.

MaxRevise' (resp. MinRevise') uses the implicit functions (without generating them explicitly) for contracting each occurrence x' of an increasing variable $x \in X$ (provided that $f_{x'}$ is monotonic increasing w.r.t. x).

Consider the same constraint of the previous example. MaxRevise' performs a slightly modified HC4-Revise in the constraint

$$4x - y^2x + x^2 - 40 \geq 0$$

[1] Recall that the method computes the interval partial derivative related to a variable as the sum of the interval partial derivatives related to each occurrence (see Section 2.2.4.2, page 16).

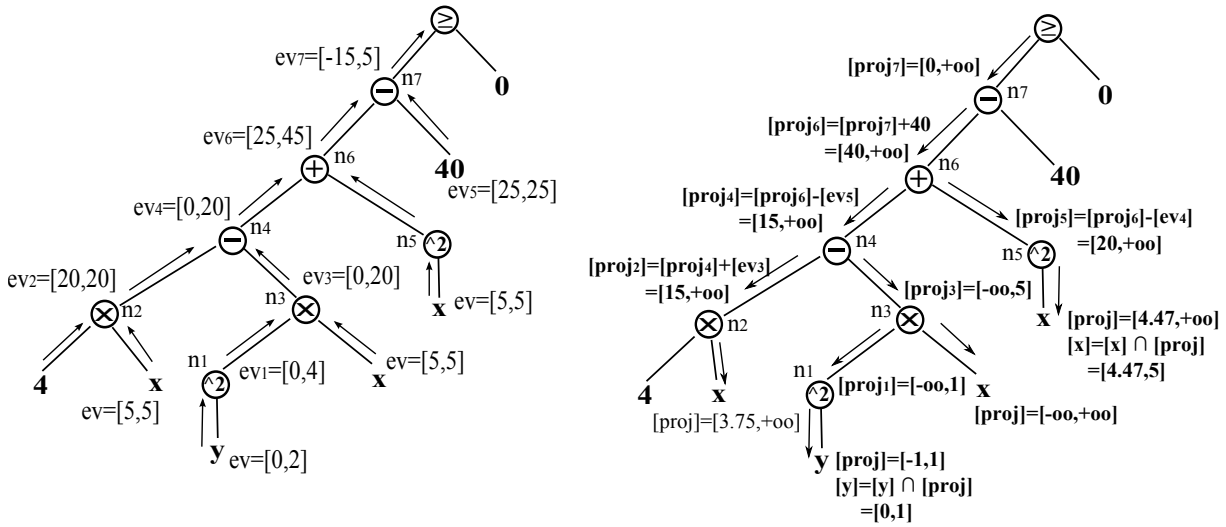

Figure 4.7: The evaluation and narrowing phases of the `MinMaxRevise'` procedure.

Each node of the expression tree maintains *two* intervals: evaluation and projection intervals. The *evaluation* intervals (ev) are computed by evaluating each node from the leaves to the root with the interval operators (see Figure 4.7-left). The monotonic increasing variables are replaced by the right bound of the interval (monotonic decreasing variables are replaced by the left bound). If the evaluation interval corresponding to the left side of the constraint (node n_7) is less than 0 the procedure certifies that there are no solution in the box.

The projection intervals are computed from the root to the leaves (see Figure 4.7-right). The projection of each node is computed using the related narrowing operator (narrowing operators are described in Section 3.2.3.3). These operators use the projection interval of the father and the evaluation interval of the brother. Consider for example the node n_5 in the figure. The contraction is performed:

$$[proj_5] \leftarrow N_{n_5}^{n_4+n_5}([proj_6], [ev_4]) = ([40, +\infty] - [0, 20]) = [20, +\infty]$$

In a similar way the contraction of the node n_4 is performed as follows:

$$[proj_4] \leftarrow N_{n_4}^{n_4+n_5}([proj_6], [ev_5]) = ([40, +\infty] - [25, 25]) = [15, +\infty]$$

Contrarily to the classical algorithm, it is not correct to intersect the evaluation of a node with the projection on the same node. Indeed, in the evaluation we used single values for the monotonic variables whereas we want to project on the entire intervals.

At the end of the narrowing phase, each interval $[x']$ (related to an occurrence of an increasing variable x) is intersected with its projection interval if the interval partial derivative of x' satisfies the condition (4.5).

In the example, the first occurrence of x (with partial derivative $[g_{x'}] = [4, 4]$) cannot be contracted because $\underline{g_x} - \overline{g_{x'}} = 2 - 4 \not\geq 0$. The second occurrence is not contracted because the projection interval contains the entire interval $[x]$. Finally, the interval related to the third occurrence is contracted to $[4.47, 5]$ because $\underline{g_x} - \overline{g_{x'}} = 2 - 2 \geq 0$ (the same result obtained by the projection of the implicit function in (4.6)).

Remark that the `MaxRevise'` procedure performs (automatically) the projections related to each occurrence of x' using its implicit function $f_{x'}$.

Consider an occurrence x' related to an interval $[x']$ and to an increasing variable x. `MinMaxRevise'` contracts $[x']$ better than or as well as `HC4-Revise` if $\underline{g_x} - \underline{g_{x'}} \geq 0$. However, if $\underline{g_x} - \underline{g_{x'}} \not\geq 0$, `MinMaxRevise'` does not contract $[x']$ whereas `HC4-Revise` might contract it.

4.8 Related Work

The algorithm `Octum`, described in Section 3.2.3.6, contains a procedure similar to `MonotonicBoxNarrow` for narrowing monotonic variables.

Inspired by `Octum` we have adapted Proposition 8 to interval functions for improving the latest version of our `MonotonicBoxNarrow` procedure. Compared to `Octum`, `Mohc` presents additional features:

- `Mohc` exploits the monotonicity when a function becomes monotonic w.r.t. *one or more* variables in the current box. `Octum` requires a function be monotonic w.r.t. *all* its variables in the current box simultaneously.
- `Mohc` uses the `Occurrence grouping` procedure (contribution presented in Chapter 5) that quickly rewrites the constraint expressions in order to detect more cases of monotonicity.
- Contrarily to `Octum`, `Mohc` uses a `HC4-Revise`-based function to contract quickly the intervals of variables occurring once and of those which are not monotonic. `Mohc` does not need to call a more costly `BoxNarrow`-based procedure to handle monotonic variables that appear once in the expression.
- `Mohc` performs a test that avoids several calls to the procedures `LeftNarrowFmin`, `RightNarrowFmax`, `LeftNarrowFmax` and `RightNarrowFmin` (see the first improvement in Section 4.4.4).

The `ALIAS` library implements a contractor performing a similar contraction that the `MinMaxRevise` and `MinMaxRevise'` procedures do. First, as a preprocessing `ALIAS` generates explicitily the projection functions related to each variable occurrence using symbolic computation tools. When the function is treated, each occurrence of every variable is contracted by using the evaluation by monotonicity of its related projection function(s). Remark that, in the same way as `MinMaxRevise` (and `MinMaxRevise'`, the contraction is not optimal because each contracted occurrence is considered as a different variable.

4.9 Conclusion and Future Work

This chapter has presented a new interval constraint propagation algorithm exploiting the monotonicity of functions. Using ingredients present in the existing `HC4-Revise` and `BoxNarrow`, `Mohc` has the potential to advantageously replace `HC4` and `Box`, as shown by our experiments.

A future work is related with the variables in W. If a function f is monotonic w.r.t. all the variables with multiple occurrences *except one* (w), then the classical `BoxNarrow` procedure (see Section 3.2.3.5) using the evaluation by monotonicity is able to project *optimally* on w. Thus, we believe that applying `BoxNarrow` over variables in W using the evaluation by monotonicity can lead to good results.

As suggested by our experiment about the application frequency of the monotonicity-based procedures (see Section 4.6.1.4), we will implement an auto-adaptive version of the τ_{mohc} parameter based on the following intuition. If the extension by monotonicity *often* computes good evaluations w.r.t. the natural

extension, then `Mohc` should apply the monotonicity-based procedures more often (i.e., τ_{mohc} should increase). The expression for qualifying these good evaluations would be ρ_{mohc} (see Section 4.3.1) and the method for changing the value of τ_{mohc} could be a reinforcement learning technique.

Finally, three points suggest the more ambitious idea of combining constraints linearly such that the monotonicity of the combination brings a significant contraction:

- The possibility of computing the hull-consistency in monotonic functions with multiple occurrences of variables.
- The reduction of the dependency problem thanks to our occurrence grouping extension (presented in Chapter 5).
- Our sophisticated algorithm `MinMaxRevise'` for computing projections over each variable occurrence using the monotonicity.

Chapter 5

A New Monotonicity-based Interval Extension

Contents

In the previous chapter, the *occurrence grouping* extension has been briefly introduced as an important procedure called by the `Mohc` algorithm. In this chapter we explain in more detail how this new interval extension works and why it computes sharper images than both the natural extension and the extension by monotonicity.

(A part of the material presented in this chapter is published in [Araya et al., 2009a].)

5.1 Introduction

(I recommend first to read carefully Section 2.4.2 for better understanding the definitions and considerations presented in this chapter.)

The computation of sharp images of functions is in the heart of interval arithmetic. As explained in Section 2.4, different interval extensions have been defined with the objective of computing sharper approximations of the optimal image.

For instance, the *natural extension* (described in Section 2.4.1) maps a function to intervals (replacing the variables by domains and arithmetic operators by interval operators) and evaluates it using interval arithmetic. The natural extension computes the optimal image when each variable occurs once in a continuous function f. A more effective extension is the *extension by monotonicity* (described in Section 2.4.2) that computes the optimal image when the function is monotonic w.r.t. all its variables. Otherwise, i.e., if the function is monotonic w.r.t. only some variables, the extension by monotonicity computes an interval sharper than the natural extension does, thanks to the monotonic variables.

The *occurrence grouping extension* presented in this chapter improves the evaluation by monotonicity related to each *non* monotonic variable x. The method consists in selecting from the occurrences of x two subgroups of occurrences, one monotonic increasing group and one monotonic decreasing one. Replacing the occurrences of each group by auxiliary variables (x_a if the occurrences are in the increasing group or x_b if they are in the decreasing one), we create a new function f^{og}. f^{og} is better evaluated by monotonicity than f.

5.2 Evaluation by monotonicity with occurrence grouping

In this section, we study the case of a function which is not monotonic w.r.t. a variable with multiple occurrences. We can, without loss of generality, limit the study to a function having a single variable: the generalization to a function having several variables is straightforward, the evaluations by monotonicity being independent.

Example 26 *Consider* $f_1(x) = -x^3+2x^2+6x$. *We want to calculate a sharp evaluation of this function when* x *falls in* $[-1.2, 1]$. *The derivative of* f_1 *is* $f_1'(x) = -3x^2+4x+6$ *and contains a positive term* (6), *a negative term* $(-3x^2)$ *and a term containing zero* $(4x)$.

$[f_1]_{opt}([B])$ *is* $[-3.05786, 7]$, *but we cannot obtain it directly by a simple interval function evaluation (one needs to solve* $f_1'(x) = 0$, *which is in the general case a problem in itself).*

In the interval $[-1.2, 1]$, *the function* f_1 *is not monotonic. The natural interval evaluation yields* $[-8.2, 10.608]$, *the Horner evaluation* $[-11.04, 9.2]$ *(see [Horner, 1819]).*

When a function f is not monotonic w.r.t. a variable x, it sometimes appears that it is monotonic w.r.t. some occurrences.

Recall that AD (see Section 2.2.4.2) computes the interval gradient of x as the sum of the interval partial derivatives of each occurrence of x, i.e.,

$$\left[\frac{\partial f}{\partial x}\right]([x]) = \sum_{i=1}^{k} \left[\frac{\partial f}{\partial x_i}\right]([x])$$

where x_i is the i^{th} occurrence of x in f. Thus, a subgroup M of occurrences such that $0 \neq \sum_{x_i \in M} \left[\frac{\partial f}{\partial x_i}\right]([x])$ is monotonic. If the occurrences in M are replaced by a new variable (x'), then the new function can be evaluated using the monotonicity of x'.

Consider the following naive grouping for Example 26. We replace the function f_1 by a function f_1^{nog}, grouping together all increasing occurrences into one variable x_a and all decreasing occurrences into one variable x_b. We obtain:

$$f_1^{nog}(x_a, x_b, x) = -x_b^3 + 2x^2 + 6x_a$$

The evaluation by monotonicity of f_1^{nog} is $[f_1^{nog}]_m([-1.2, 1]) = [-8.2, 10.608]$.

As stated in Proposition 2.4.1, page 23, the natural extension of the function f^{nog} always computes the same result as the evaluation by monotonicity. Indeed, when a node in the evaluation tree corresponds to an increasing function w.r.t. a variable occurrence, the natural evaluation automatically selects the right bound (among both) of the occurrence domain during the evaluation process.

The main idea is then to change this grouping in order to reduce the dependency problem and obtain sharper evaluations. We can in fact group some occurrences (increasing, decreasing, or non monotonic) into an increasing variable x_a as long as the function remains increasing w.r.t. this variable x_a. The evaluation will be the same or sharper.

Also, if it is possible to transfer all decreasing occurrences into the increasing part, the dependency problem will now occur only on the occurrences in the increasing and non monotonic parts.

For f_1, if we group together the positive derivative term (i.e., the third occurrence of x with derivative $[6, 6]$) with the derivative term containing zero (i.e., the second occurrence of x with derivative $[-2.4, 2]$) we obtain the new function (with sum of derivatives $[6, 6] + [-2.4, 2] = [4, 10.2]$ that still does not contain zero) :

$$f_1^{og_1}(x_a, x_b) = -x_b^3 + 2x_a^2 + 6x_a$$

where $f_1^{og_1}$ is increasing w.r.t. x_a (the interval partial derivative of x_a is in fact $[4, 10.2]$, the sum of the interval partial derivatives of its occurrences) and decreasing w.r.t. x_b. We can then use the evaluation by monotonicity for obtaining the interval $[-5.32, 9.728]$. We can in the same manner obtain

$$f_1^{og_2}(x_a, x_c) = -x_a^3 + 2x_c^2 + 6x_a$$

The evaluation by monotonicity of $f_1^{og_2}$ yields $[-5.472, 7.88]$. We remark that we find sharper images than the natural evaluation of f_1 does.

In Section 5.3, we present a linear program to perform *occurrence grouping* automatically.

Interval extension by occurrence grouping

Consider the function $f(x)$ with multiple occurrences of x. We obtain a function $f^{og}(x_a, x_b, x_c)$ by replacing in f every occurrence of x by one of the three variables x_a, x_b, x_c, such that f^{og} is increasing w.r.t. x_a in $[x]$, and f^{og} is decreasing w.r.t. x_b in $[x]$. Then, we define the *interval extension by occurrence grouping* of f by: $[f]_{og}([x]) := [f^{og}]_m([x], [x], [x])$

Unlike the natural interval extension and the interval extension by monotonicity, the interval extension by occurrence grouping is not unique for a function f since it depends on the occurrence grouping (og) that transforms f into f^{og}.

5.3 A 0,1 linear program to perform occurrence grouping

In this section, we propose a method for automatizing occurrence grouping. Using the Taylor extension, we first compute an over-estimate of the diameter of the image computed by $[f]_{og}$. Then, we propose a linear program performing a grouping that minimizes this over-estimate.

5.3.1 Taylor-based over-estimate

On one hand, x_c has not been detected monotonic, and the evaluation by monotonicity considers the occurrences of x_c as different variables such as the natural evaluation would. On the other hand, as f^{og} is monotonic w.r.t. x_a and x_b, the evaluation by monotonicity of these variables is optimal. Proposition 4 has been introduced in Section 2.4.2, page 25. It says that the evaluation by monotonicity computes the optimal image of a monotonic, continuous and differentiable function. Proposition 14 is a straightforward corollary of Proposition 2 introduced in Section 2.4.1.

Proposition 14 *Let $f(x)$ be a continuous function in the interval $[x]$ with k occurrences of x. $f^\circ(x_1,..,x_k)$ is a function obtained from f by considering all the occurrences of x as different variables.*

Then, $[f]_n([x])$ computes $[f^\circ]_{opt}([x],...,[x])$.

Using Proposition 4 and Proposition 14, we observe that the evaluation by monotonicity of $f^{og}(x_a,x_b,x_c)$ is equivalent to the optimal evaluation of $f^\circ(x_a,x_b,x_{c_1},...,x_{c_{ck}})$, considering each occurrence of x_c in f^{og} as an independent variable x_{c_j} in f°.

Proposition 15 computes an upper bound of $\texttt{Diam}([f]_{opt}([B]))$ deduced from the Taylor extension (see Definition 5 of Section 2.4.3) and basic interval diameter properties (see Section 2.2.1).

Proposition 15 *Let $f(x_1,...,x_n)$ be a function with domains $[B]=[x_1]\times...\times[x_n]$. Then,*

$$\texttt{Diam}([f]_{opt}([B])) \leq \sum_{i=1}^{n}\left(\texttt{Diam}([x_i])\times\left|\left[\frac{\partial f}{\partial x_i}\right]([B])\right|\right) \tag{5.1}$$

Using Proposition 15, we can calculate an upper bound of the **diameter** of $[f]_{og}([B])=[f^{og}]_m([B])=[f^\circ]_{opt}([B])$[1]:

$$\texttt{Diam}([f]_{og}([B])) \leq \texttt{Diam}([x])\left(\left|\left[\frac{\partial f^{og}}{\partial x_a}\right]([B])\right|+\left|\left[\frac{\partial f^{og}}{\partial x_b}\right]([B])\right|+\sum_{i=1}^{ck}\left|\left[\frac{\partial f^{og}}{\partial x_{c_i}}\right]([B])\right|\right)$$

In order to respect the monotonicity conditions required by f^{og}: $\left[\frac{\partial f^{og}}{\partial x_a}\right]_{opt}([B])\geq 0$, $\left[\frac{\partial f^{og}}{\partial x_b}\right]_{opt}([B])\leq 0$, we have the sufficient conditions $\underline{\left[\frac{\partial f^{og}}{\partial x_a}\right]([B])}\geq 0$ and $\overline{\left[\frac{\partial f^{og}}{\partial x_b}\right]([B])}\leq 0$, implying $\left|\left[\frac{\partial f^{og}}{\partial x_a}\right]([B])\right| = \overline{\left[\frac{\partial f^{og}}{\partial x_a}\right]([B])}$ and $\left|\left[\frac{\partial f^{og}}{\partial x_b}\right]([B])\right| = -\underline{\left[\frac{\partial f^{og}}{\partial x_b}\right]([B])}$. Finally:

$$\texttt{Diam}([f]_{og}([B])) \leq \texttt{Diam}([x])\left(\overline{\left[\frac{\partial f^{og}}{\partial x_a}\right]([B])}-\underline{\left[\frac{\partial f^{og}}{\partial x_b}\right]([B])}+\sum_{i=1}^{ck}\left|\left[\frac{\partial f^{og}}{\partial x_{c_i}}\right]([B])\right|\right) \tag{5.2}$$

5.3.2 A linear program

We want to transform f into a new function f^{og} that minimizes the right side of the relation (5.2). The problem can be easily transformed into the following integer linear program:

Find the values r_{a_i}, r_{b_i} and r_{c_i} for each occurrence x_i that minimize

$$G=\overline{g_a}-\underline{g_b}+\sum_{i=1}^{k}\left(|[g_i]|\,r_{c_i}\right) \tag{5.3}$$

subject to:

$$\underline{g_a}\geq 0 \tag{5.4}$$

[1]For simplicity we denote $[B]$ the box $[x]\times...\times[x]$ containing as many intervals $[x]$ as required by the interval function. For instance in $[f]_{og}([B])$, $[B]=[x]$; in $[f^{og}]_m([B])$, $[B]=[x]\times[x]\times[x]$.

$$\overline{g_b} \leq 0 \tag{5.5}$$

$$\begin{aligned} r_{a_i} + r_{b_i} + r_{c_i} &= 1 \quad \text{for } i = 1, ..., k \\ r_{a_i}, r_{b_i}, r_{c_i} &\in \{0, 1\} \quad \text{for } i = 1, ..., k, \end{aligned} \tag{5.6}$$

where:

$$\begin{aligned} [g_a] &= \left[\frac{\partial f^{og}}{\partial x_a}\right]([B]) = \sum_{i=1}^{k} [g_i]\, r_{a_i} \\ [g_b] &= \left[\frac{\partial f^{og}}{\partial x_b}\right]([B]) = \sum_{i=1}^{k} [g_i]\, r_{b_i} \\ [g_i] &= \left[\frac{\partial f^{og}}{\partial x_i}\right]([B]) \end{aligned}$$

k is the number of occurrences of x. A value r_{a_i}, r_{b_i} or r_{c_i} equal to 1 indicates that the occurrence x_i in f will be replaced, respectively, by x_a, x_b or x_c in f^{og}.

We can remark that the interval partial derivatives related to the auxiliary variables (i.e., $[g_a]$, $[g_b]$) are calculated by using only the interval partial derivatives related to each occurrence of x[1].

Linear program corresponding to Example 26

We have $f_1(x) = -x^3 + 2x^2 + 6x$, $f_1'(x) = -3x^2 + 4x + 6$ for $x \in [-1.2, 1]$. The interval derivative values for each occurrence are: $[g_1] = [-4.32, 0]$, $[g_2] = [-4.8, 4]$ and $[g_3] = [6, 6]$. Then, the linear program is:

Find the values r_{a_i}, r_{b_i} and r_{c_i} that minimize

$$\begin{aligned} G &= \sum_{i=1}^{3} \overline{g_i}\, r_{a_i} - \sum_{i=1}^{3} \underline{g_i}\, r_{b_i} + \sum_{i=1}^{3} \left(|[g_i]|\, r_{c_i}\right) \\ &= (4r_{a_2} + 6r_{a_3}) + (4.32r_{b_1} + 4.8r_{b_2} - 6r_{b_3}) + (4.32r_{c_1} + 4.8r_{c_2} + 6r_{c_3}) \end{aligned}$$

subject to:

$$\sum_{i=1}^{3} \underline{g_i}\, r_{a_i} = -4.32r_{a_1} - 4.8r_{a_2} + 6r_{a_3} \geq 0$$

$$\sum_{i=1}^{3} \overline{g_i}\, r_{b_i} = 4r_{b_2} + 6r_{b_3} \leq 0$$

$$\begin{aligned} r_{a_i} + r_{b_i} + r_{c_i} &= 1 \quad \text{for } i = 1, ..., 3 \\ r_{a_i}, r_{b_i}, r_{c_i} &\in \{0, 1\} \quad \text{for } i = 1, ..., 3 \end{aligned}$$

[1]This implies that, even if the interval partial derivatives related to the occurrences are optimally calculated, the interval derivatives of x_a and x_b can suffer from the dependency problem because each occurrence derivative is computed independently.

We obtain the minimum 10.8, and the solution $r_{a_1} = 1, r_{b_1} = 0, r_{c_1} = 0, r_{a_2} = 0, r_{b_2} = 0, r_{c_2} = 1, r_{a_3} = 1, r_{b_3} = 0, r_{c_3} = 0$. We can remark that the value of the over-estimate of $\texttt{Diam}([f]_{og}([B]))$ is equal to 23.76 ($10.8 \times \texttt{Diam}([-1.2, 1])$) whereas $\texttt{Diam}([f]_{og}([B])) = 13.352$. Although the over-estimate is quite rough, the heuristic works well on this example. Indeed, $\texttt{Diam}([f]_n([B])) = 18.808$, and $\texttt{Diam}([f]_{opt}([B])) = 10.06$.

5.4 A linear programming problem achieving a better occurrence grouping

The linear program above is a 0,1 linear program and is known to be NP-hard in general. We can render it tractable while, more important in practice, improving the minimum G by allowing r_{a_i}, r_{b_i} and r_{c_i} to get *real* values. In other words, we allow each occurrence of x in f to be replaced by a *convex linear combination* of auxiliary variables, x_a, x_b and x_c such that f^{og} is increasing w.r.t. x_a and decreasing w.r.t. x_b in $[x]$.

Definition 18 *(**Interval extension by occurrence grouping**)*
Let $f(x)$ be a function with multiple occurrences of the variable x. $f^{og}(x_a, x_b, x_c)$ is the function obtained by replacing in f every occurrence of x by $r_{a_i}x_a + r_{b_i}x_b + r_{c_i}x_c$, such that:

- $r_{a_i}, r_{b_i}, r_{c_i} \in [0,1]^3$ *and* $r_{a_i} + r_{b_i} + r_{c_i} = 1$,
- $\frac{\partial f^{og}}{\partial x_a}([x],[x],[x]) \geq 0$ *and* $\frac{\partial f^{og}}{\partial x_b}([x],[x],[x]) \leq 0$.

The interval extension by occurrence grouping of f is defined by $[f]_{og}([x]) := [f^{og}]_m([x],[x],[x])$

In Example 26, we can replace f_1 by f^{og_1} or f^{og_2} in a way respecting the monotonicity constraints of x_a and x_b:

1. $f_1^{og_1}(x_a, x_b) = -(\frac{5}{18}x_a + \frac{13}{18}x_b)^3 + 2x_a^2 + 6x_a$: $[f_1^{og_1}]_m([x]) = [-4.38, 8.205]$
2. $f_1^{og_2}(x_a, x_b, x_c) = -x_a^3 + 2(0.35x_a + 0.65x_c)^2 + 6x_a$: $[f_1^{og_2}]_m([x]) = [-5.472, 7]$

Example 27 *Consider the function $f_2(x) = x^3 - x$ and the interval $[x] = [0.5, 2]$. f_2 is not monotonic and the optimal image $[f_2]_{opt}([x])$ is $[-0.385, 6]$.*
The natural evaluation yields $[-1.975, 7.5]$, the Horner evaluation $[-1.5, 6]$. We can replace f_2 by one of the following functions:

1. $f_2^{og_1}(x_a, x_b) = x_a^3 - (\frac{1}{4}x_a + \frac{3}{4}x_b)$: $[f_2^{og_1}]_m([x]) = [-0.75, 6.375]$
2. $f_2^{og_2}(x_a, x_b) = (\frac{11}{12}x_a + \frac{1}{12}x_b)^3 - x_b$: $[f_2^{og_2}]_m([x]) = [-1.756, 6.09]$

Thus, the new linear program that computes convex linear combinations for achieving occurrence grouping becomes:

Find the values r_{a_i}, r_{b_i} and r_{c_i} for each occurrence x_i that minimize (5.3) subject to (5.4), (5.5), (5.6) and

$$r_{a_i}, r_{b_i}, r_{c_i} \in [0,1] \quad \text{for } i = 1, ..., k. \tag{5.7}$$

Note that this continuous linear program improves the minimum of the objective function because the integrity conditions are relaxed.

Linear program corresponding to Example 26

In this example we obtain the minimum 10.58 and the new function

$$f_1^{og}(x_a, x_b, x_c) = -x_a^3 + 2(0.35x_a + 0.65x_c)^2 + 6x_a : \quad [f_1^{og}]_m([x]) = [-5.472, 7]$$

The minimum 10.58 is less than 10.8 (minimum obtained by the 0,1 linear program). The evaluation by occurrence grouping of f_1 yields $[-5.472, 7]$, which is sharper than the image $[-5.472, 7.88]$ obtained by the 0.1 linear program presented in Section 5.3.

Linear program corresponding to Example 27

In this example, we obtain the minimum 11.25 and the new function

$$f_2^{og}(x_a, x_b) = \left(\frac{44}{45}x_a + \frac{1}{45}x_b\right)^3 - \left(\frac{11}{15}x_a + \frac{4}{15}x_b\right)$$

The image $[-0.75, 6.01]$ obtained by occurrence grouping is sharper than the interval computed by natural and Horner evaluations. Note that in this case the 0,1 linear program of Section 5.3 yields the same grouping as the naive strategy presented in Section 5.2.

Thus, the continuous linear program not only makes the problem tractable but also improves the minimum of the objective function.

5.5 An efficient Occurrence Grouping algorithm

Algorithm 11 `Occurrence_Grouping`(**in:** f, $[g_*]$ **out:** f^{og})

1: $[G_0] \leftarrow \sum_{i=1}^{k} [g_i]$
2: $[G_m] \leftarrow \sum_{0 \notin [g_i]} [g_i]$
3: **if** $0 \notin [G_0]$ **then**
4: `OG_case1`$([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
5: **else if** $0 \in [G_m]$ **then**
6: `OG_case2`$([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
7: **else**
8: /* $0 \notin [G_m]$ and $0 \in [G_0]$ */
9: **if** $\underline{G_m} \geq 0$ **then**
10: `OG_case3`$^+([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
11: **else**
12: `OG_case3`$^-([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
13: **end if**
14: **end if**
15: $f^{og} \leftarrow$ `Generate_New_Function`$(f, [r_{a_*}], [r_{b_*}], [r_{c_*}])$

Algorithm 11 finds r_{a_i}, r_{b_i}, r_{c_i} (r-values) that minimize G subject to the constraints. At line 15, the algorithm generates symbolically the new function f^{og} that replaces each occurrence x_i in f by $[r_{a_i}]x_a +$

$[r_{b_i}]x_b + [r_{c_i}]x_c$. Note that the values are represented by thin intervals, of a few u.l.p. large, for taking into account the floating point rounding errors appearing in the computations.

Algorithm 11 uses a vector $[g_*]$ of size k containing interval derivatives of f w.r.t. *each occurrence* x_i of x. Each component of $[g_*]$ is denoted by $[g_i]$ and corresponds to the interval $\left[\frac{\partial f}{\partial x_i}\right]([B])$. A symbol indexed by an asterisk refers to a vector (e.g., $[g_*]$, $[r_{a_*}]$).

We illustrate the algorithm using the two univariate functions of our examples: $f_1(x) = -x^3 + 2x^2 + 6x$ and $f_2(x) = x^3 - x$ for domains of x: $[-1.2, 1]$ and $[0.5, 2]$ respectively. The interval derivatives of f w.r.t. each occurrence of x have been previously calculated. For the examples, the interval derivatives of f_2 w.r.t. x occurrences are $[g_1] = [0.75, 12]$ and $[g_2] = [-1, -1]$; the interval derivatives of f_1 w.r.t. x occurrences are $[g_1] = [-4.32, 0]$, $[g_2] = [-4.8, 4]$ and $[g_3] = [6, 6]$.

At line 1, the partial derivative $[G_0]$ of f w.r.t. x is calculated using the sum of the partial derivatives of f w.r.t. each occurrence of x. In line 2, $[G_m]$ gets the value of the partial derivative of f w.r.t. the monotonic occurrences of x. In the examples, for f_1: $[G_0] = [g_1] + [g_2] + [g_3] = [-3.12, 10]$ and $[G_m] = [g_1] + [g_3] = [1.68, 6]$, and for f_2: $[G_0] = [G_m] = [g_1] + [g_2] = [-0.25, 11]$.

According to the values of $[G_0]$ and $[G_m]$, we can distinguish 3 cases. The first case is well-known ($0 \notin [G_0]$ in line 3) and occurs when x is a monotonic variable. The procedure `OG_case1` does not achieve any occurrence grouping: *all the occurrences* of x are replaced by x_a (if $[G_0] \geq 0$) or by x_b (if $[G_0] \leq 0$). The evaluation by monotonicity of f^{og} is equivalent to the evaluation by monotonicity of f.

In the second case, when $0 \in [G_m]$ (line 5), the procedure `OG_case2` (Algorithm 12) achieves a grouping of the occurrences of x. Increasing occurrences are replaced by $(1-\alpha_1)x_a + \alpha_1 x_b$, decreasing occurrences by $\alpha_2 x_a + (1-\alpha_2)x_b$ and non monotonic occurrences by x_c (lines 7 to 13 of Algorithm 12). f_2 falls in this case: $\alpha_1 = \frac{1}{45}$ and $\alpha_2 = \frac{11}{15}$ are calculated in lines 3 and 4 of Algorithm 12 using $[G^+] = [g_1] = [0.75, 12]$ and $[G^-] = [g_2] = [-1, -1]$. The new function becomes: $f_2^{og}(x_a, x_b) = (\frac{44}{45}x_a + \frac{1}{45}x_b)^3 - (\frac{11}{15}x_a + \frac{4}{15}x_b)$.

Algorithm 12 `OG_case2`(**in:** $[g_*]$ **out:** $[r_{a_*}], [r_{b_*}], [r_{c_*}]$)

1: $[G^+] \leftarrow \sum_{[g_i] \geq 0} [g_i]$
2: $[G^-] \leftarrow \sum_{[g_i] \leq 0} [g_i]$
3: $[\alpha_1] \leftarrow \dfrac{\underline{G^+}\overline{G^-} + \overline{G^-}\underline{G^-}}{\underline{G^+}\overline{G^-} - \underline{G^-}\overline{G^+}}$
4: $[\alpha_2] \leftarrow \dfrac{\underline{G^+}\overline{G^+} + \overline{G^-}\underline{G^+}}{\underline{G^+}\overline{G^-} - \underline{G^-}\overline{G^+}}$
5:
6: **for all** $[g_i] \in [g_*]$ **do**
7: **if** $\underline{g_i} \geq 0$ **then**
8: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (1 - [\alpha_1], [\alpha_1], 0)$
9: **else if** $\overline{g_i} \leq 0$ **then**
10: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow ([\alpha_2], 1 - [\alpha_2], 0)$
11: **else**
12: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (0, 0, 1)$
13: **end if**
14: **end for**

The third case occurs when $0 \notin [G_m]$ and $0 \in [G_0]$. W.l.o.g., assume that $\underline{G_m} \geq 0$. The procedure `OG_case3`$^+$ (Algorithm 13) first groups all the positive and negative occurrences in the increasing group

Algorithm 13 `OG_case3`$^+$(**in:**$[g_*]$ **out:**$[r_{a_*}], [r_{b_*}], [r_{c_*}]$)

1: $[g_a] \leftarrow [0,0]$
2: **for all** $[g_i] \in [g_*]$, $\underline{g_i} \geq 0$ **or** $\overline{g_i} \leq 0$ **do**
3: $\quad [g_a] \leftarrow [g_a] + [g_i]$ /* All positive and negative derivatives are absorbed by $[g_a]$ */
4: $\quad ([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (1,0,0)$
5: **end for**
6:
7: $index \leftarrow$ `descending_sort` $(\{[g_i] \in [g_*], \underline{g_i} < 0\}, \texttt{criterion} \rightarrow \frac{\overline{g_i} - |[g_i]|}{\underline{g_i}})$
8: $j \leftarrow 1$; $i \leftarrow index[1]$
9: **while** $\underline{g_a} + \underline{g_i} \geq 0$ **do**
10: $\quad ([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (1,0,0)$
11: $\quad [g_a] \leftarrow [g_a] + [g_i]$
12: $\quad j \leftarrow j+1$; $i \leftarrow index[j]$
13: **end while**
14:
15: $[\alpha] \leftarrow -\frac{\underline{g_a}}{\underline{g_i}}$
16: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow ([\alpha], 0, 1-[\alpha])$ /* $[g_a] \leftarrow [g_a] + [\alpha][g_i]$ */
17:
18: $j \leftarrow j+1$; $i \leftarrow index[j]$
19: **while** $j \leq$ `length`$(index)$ **do**
20: $\quad ([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (0,0,1)$
21: $\quad j \leftarrow j+1$; $i \leftarrow index[j]$
22: **end while**

x_a (lines 2–5). The non monotonic occurrences are then replaced by x_a in an order determined by an array $index$[1] (line 7) as long as the constraint $\sum_{i=1}^{k} r_{a_i}\underline{g_i} \geq 0$ is satisfied (lines 9–13). The criterion varies from 0 (for non monotonic occurrences having $|[g_i]| = \overline{g_i}$) to 1^- (for occurrences having $\overline{g_i} = 0^+$). The first occurrence $x_{i'}$ that cannot be (entirely) replaced by x_a because it would make the constraint (5.4) unsatisfiable is replaced by $\alpha x_a + (1-\alpha)x_c$, with α such that the constraint is satisfied and equal to 0, i.e., $(\sum_{i=1, i\neq i'}^{k} r_{a_i}\underline{g_i}) + \alpha\underline{g_{i'}} = 0$ (lines 15–16). The rest of the occurrences are replaced by x_c (lines 18–22). f_1 falls in this case. The increasing and decreasing occurrences of x are first replaced by x_a.The second occurrence of x, that is non monotonic, is then replaced by $\alpha x_a + (1-\alpha)x_c$, where $\alpha = 0.35$ is obtained by forcing the constraint (5.4) to be 0: $\underline{g_1} + \underline{g_3} + \alpha\underline{g_2} = 0$. The new function is: $f_1^{og}(x_a, x_b, x_c) = -x_a^3 + 2(0.35x_a + 0.65x_c)^2 + 6x_a$.

5.5.1 Properties

Proposition 16 *Algorithm 11 (*`Occurrence_grouping`*) is correct, i.e., it is an interval extension of f.*

Proposition 16 implies that Algorithm 11 respects the four constraints (5.4)–(5.7). A full proof of Proposition 16 can be found in Section B.1.

[1] An occurrence x_{i_1} is handled before x_{i_2} if $\frac{\overline{g_{i1}} - |[g_{i1}]|}{\underline{g_{i1}}} \geq \frac{\overline{g_{i2}} - |[g_{i2}]|}{\underline{g_{i2}}}$. $index[j]$ yields the index of the j^{th} occurrence in this order.

Proposition 17 *Let* $[g_i]$ *be the intervals* $\left[\frac{\partial f^{og}}{\partial x_i}\right]([B])$ *(*$i = 1...k$*). If* $0 \in [G_m] = \sum_{i=1..k, 0 \notin [g_i]} [g_i]$*, then Algorithm 12 finds the values* r_{a_i}*,* r_{b_i} *and* r_{c_i} *for all* i *that minimize (5.3) subject to (5.4), (5.5), (5.6) and (5.7).*

Proposition 18 *Let* $[g_i]$ *be the intervals* $\left[\frac{\partial f^{og}}{\partial x_i}\right]([B])$ *(*$i = 1...k$*). If* $\underline{G_m} \geq 0$ *(resp.* $\overline{G_m} \leq 0$*) and* $0 \in [G_0] = \sum_{i=1}^{k} [g_i]$ *(with* $[G_m] = \sum_{i=1..k, 0 \notin [g_i]} [g_i]$*). Then, the algorithm* `OG_case3`$^+$ *(resp.* `OG_case3`$^-$*) finds the values* r_{a_i}*,* r_{b_i} *and* r_{c_i} *for all* i *that minimize (5.3) subject to (5.4), (5.5), (5.6) and (5.7).*

Propositions 17 and 18 show that Algorithm 11 reachs the minimum of the objective function (5.3). The proof concerning Algorithm 13 (`OG_case3`) is sophisticated, due to the sort of indices, and uses known results about the continuous knapsack problem. Special care has been brought to ensure the correctness modulo floating-point roundings. Full proofs of both propositions can be found in sections B.2 and B.3 respectively.

Proposition 19 *The time complexity of* `Occurrence_Grouping` *for one variable with* k *occurrences is* $O(k\,log_2(k))$*. It is time* $O(n\,k\,log_2(k))$ *when a multi-variate function is iteratively transformed by* `Occurrence_Grouping` *for each of its* n *variables having at most* k *occurrences each.*

(A preliminary gradient calculation by automatic differentiation is time $O(e)$*, where* e *is the number of unary and binary operators in the expression.)*

The time complexity of Algorithm 11 is dominated by that of `descending_sort` in the `OG_case3` procedure.

Instead of Algorithm 11, we may use a standard Simplex algorithm providing that the used Simplex implementation takes into account floating-point rounding errors. A comparison of respective performances of Algorithm 11 and Simplex is shown in Section 5.6.3. Also, as shown in Section 5.6.1, the time required in practice by `Occurrence_Grouping` is negligible when it is used for solving systems of equations.

Although `Occurrence_Grouping` can be viewed as a heuristic since it minimizes a Taylor-based overestimate of the function image diameter, it is important to stress that our new interval extension improves the well-known monotonicity-based interval extension.

Proposition 20 *Consider a function* $f : \mathbb{R}^n \to \mathbb{R}$*, and the previously defined interval natural (*$[f]_n$*), monotonicity-based (*$[f]_m$*) and occurrence grouping (*$[f]_{og}$*) extensions of* f*. Let* X *be the* n *variables involved in* f *with domains* $[B]$*. Then,* $[f]_{og}([B]) \subseteq [f]_m([B]) \subseteq [f]_n([B])$

5.6 Experiments

`Occurrence_Grouping` has been implemented in the `Ibex` [Chabert, 2009; Chabert and Jaulin, 2009a] open source interval-based solver in `C++`. The goal of these experiments is to show the improvements in CPU time brought by `Occurrence_Grouping` when solving systems of equations. Sixteen benchmarks are issued from the COPRIN website [Merlet, 2009]. They correspond to square systems with a finite number of zero-dimensional solutions of at least two constraints involving multiple occurrences of variables and requiring more than 1 second to be solved (considering the times appearing in the website).

5.6.1 Occurrence grouping for improving a monotonicity-based existence test

First, `Occurrence_Grouping` has been implemented to be used in a monotonicity-based existence test (`OG` in Table 5.1), i.e., an occurrence grouping transforming f into f^{og} is applied to all the functions after a bisection and before a contraction. Then, a monotonicity-based existence test is applied to every produced f^{og}: if the evaluation by monotonicity of any f^{og} does not contain 0, then the current box is eliminated.

The competitor (¬`OG`) directly applies the monotonicity-based existence test to f without occurrence grouping. The contractors used in both cases are the same: `3BCID` [Trombettoni and Chabert, 2007] and Interval Newton.

The first and fifth columns of Table 5.1 indicate the name of each instance, the second and sixth columns yield the CPU time (above) and the number of nodes (below) required by a strategy based on `3BCID` (i.e., with no existence test) on an `Intel 6600 2.4 GHz`. The third and seventh columns report the results obtained by the strategy using a (standard) monotonicity-based existence test followed by `3BCID`. Finally, the fourth and eighth columns report the results of our strategy using an existence test based on occurrence grouping and `3BCID`.

System	3BCID	¬OG	OG	System	3BCID	¬OG	OG
Brent 10 1008	18.9 3941	19.5 3941	19.1 3941	ButcherA 8 3	>1 day	>1 day	>1 day
Caprasse 4 18	2.51 1305	2.56 1301	2.56 1301	Fourbar 4 3	13576 8685907	6742 4278767	**1091** 963113
Hayes 8 1	39.5 17701	41.1 17701	40.7 17701	Geneig 6 10	593 205087	511 191715	**374** 158927
I5 10 30	55.0 10645	56.3 10645	56.7 10645	Pramanik 3 2	100 124661	66.6 98971	**37.2** 69271
Katsura 12 7	74.1 4317	74.5 4317	75.0 4317	Trigexp2 11 0	82.5 14287	87.0 14287	86.7 14287
Kin1 6 16	1.72 85	1.77 85	1.77 85	Trigo1 10 9	152 2691	155 2691	156 2691
Eco9 9 16	12.7 6203	13.5 6203	13.2 6203	Virasoro 8 224	7173 2.5e+6	7212 2.5e+6	7150 2.4e+6
Redeco8 8 8	5.61 2295	5.71 2295	5.66 2295	Yamamura1 8 7	9.67 2883	10.04 2883	9.86 2883

Table 5.1: Experimental results using the monotonicity-based existence test.

From these first results we can observe that only in three benchmarks `OG` is clearly better than ¬`OG` (`Fourbar`, `Geneig` and `Pramanik`). In the other ones, the evaluation by occurrence grouping seems to be useless. Indeed, in most of the benchmarks, the existence test based on occurrence grouping does not cut branches in the search tree. However, note that it does not require additional time w.r.t. ¬`OG`. This clearly shows that the time required by `Occurrence_Grouping` is negligible.

5.6.2 Occurrence grouping inside `Mohc`

(The constraint propagation algorithm `Mohc` is described in Chapter 4.)

Table 5.2 shows the results obtained by `Mohc` (see Chapter 4) without the `OG` algorithm (¬`OG`), and with `Occurrence_Grouping` (`OG`). The first and fifth columns indicate the name of each instance, the second

and sixth columns report the results obtained by the strategy using `3BCID(Mohc)` without `OG`. The third and seventh columns report the results of our strategy using `3BCID(OG+Mohc)`. The fourth and eighth columns indicate the number of calls to `Occurrence_Grouping`.

System	Mohc			System	Mohc		
	¬OG	OG	#OG calls		¬OG	OG	#OG calls
Brent	20	20.3		ButcherA	>1 day	**1722**	
10 1008	3811	3805	30867	8 3		288773	16772045
Caprasse	2.57	2.71		Fourbar	4277	**385**	
4 18	1251	867	60073	4 3	1069963	57377	8265730
Hayes	17.62	17.45		Geneig	328	**111**	
8 1	4599	4415	5316	6 10	76465	13705	2982275
I5	57.25	58.12		Pramanik	67.98	**21.23**	
10 30	10399	9757	835130	3 2	51877	12651	395083
Katsura	100	103		Trigexp2	90.5	88.2	
12 7	3711	3625	39659	11 0	14299	14301	338489
Kin1	1.82	1.79		Trigo1	137	**57.1**	
6 16	85	83	316	10 9	1513	443	75237
Eco9	13.31	13.96		Virasoro	6790	**901**	
9 16	6161	6025	70499	8 24	619471	38389	5633140
Redeco8	5.98	6.12		Yamamura1	11.59	**2.15**	
8 8	2285	2209	56312	8 7	2663	343	43589

Table 5.2: Experimental results using `Mohc`.

We observe that, for 7 of the 16 benchmarks, `Occurrence_Grouping` is able to improve the results of `Mohc`; in `ButcherA`, `Fourbar`, `Virasoro` and `Yamamura1` the gains in CPU time ($\frac{\neg OG}{OG}$) obtained are 30, 11, 5.6 and 5.4 respectively.

The percentage of time required for `Occurrence_Grouping` w.r.t. the total solving time is 11% in `Virasoro`, 9% in `Fourbar`, 7% in `Pramanik`, 5% in `Geneig` and 3% or less in the other benchmarks. Details appear in Table 4.2, page 89.

Table 4.3, page 91, indicates that OG takes between 2% and 85% (34% on average) the time of one call to `HC4-Revise`. This overhead is negligible over the gradient calculation (116% on average) plus the two natural evaluations ($\sim$ 100%) used in a standard evaluation by monotonicity.

5.6.3 Performance comparison with Simplex

We have compared the performance of two Occurrence Grouping implementations: using our ad-hoc algorithm (`Occurrence_Grouping`) and using a Simplex method (`Simplex_Occurrence_Grouping`).[1]

Two important results have been obtained: first, we have checked experimentally that our algorithm is correct, i.e., it obtains the minimum value for the objective function G. Second, just as we expected, the performance of `Simplex_Occurrence_Grouping` is worse than the performance of our algorithm taking, in average, between 2.32 (`Brent`) and 10 (`Virasoro`) times more time.

[1] The Simplex algorithm has been adapted from `pagesperso-orange.fr/jean-pierre.moreau/Cplus/tsimplex_cpp.txt`. It is not rigorous, i.e., it does not take into account rounding errors due to floating point arithmetic. Adding this feature should make the algorithm work even more slowly.

5.6.4 Evaluation diameter comparison

Table 5.3 reports a comparison between the evaluation by occurrence grouping ($[f]_{og}$) and a set of interval evaluations including Taylor (see Section 2.4.3), Hansen (see Section 2.4.4) and monotonicity-based extensions (see Section 2.4.2). The first column indicates the name of each instance. The other columns are related to different extensions $[f]_{ext}$ and report the average[1] of the ratios $\rho_{ext} = \frac{Diam([f]_{og})}{Diam([f]_{ext})}$ calculated, using the `Mohc` algorithm, in each revise procedure of a function f.

NCSP	$[f]$	$[f]_t$	$[f]_h$	$[f]_m$	$[f]_{mr}$	$[f]_{mr+h}$	$[f]_{mr+og}$
`Brent`	0.857	0.985	0.987	0.997	0.998	0.999	1.000
`ButcherA`	0.480	0.742	0.863	0.666	0.786	0.963	1.028
`Caprasse`	0.602	0.883	0.960	0.856	0.953	1.043	1.051
`Direct kin.`	0.437	0.806	0.885	0.875	0.921	0.979	1.017
`Eco9`	0.724	0.785	0.888	0.961	0.980	0.976	1.006
`Fourbar`	0.268	0.718	0.919	0.380	0.427	1.040	1.038
`Geneig`	0.450	0.750	0.847	0.823	0.914	0.971	1.032
`Hayes`	0.432	0.966	0.974	0.993	0.994	0.998	1.001
`I5`	0.775	0.859	0.869	0.925	0.932	0.897	1.005
`Katsura`	0.620	0.853	0.900	0.993	0.999	0.999	1.000
`Kin1`	0.765	0.872	0.880	0.983	0.983	0.995	1.001
`Pramanik`	0.375	0.728	0.837	0.666	0.689	0.929	1.017
`Redeco8`	0.665	0.742	0.881	0.952	0.972	0.997	1.011
`Trigexp2`	0.904	0.904	0.904	0.942	0.945	0.921	1.002
`Trigo1`	0.483	0.766	0.766	0.814	0.814	0.895	1.000
`Virasoro`	0.479	0.738	0.859	0.781	0.795	1.025	1.062
`Yamam.1`	0.272	0.870	0.870	0.758	0.758	0.910	1.000
AVERAGE	0.564	0.822	0.888	0.845	0.874	0.973	1.016

Table 5.3: Different evaluations compared to $[f]_{og}$

The list of interval extensions and their related columns in the table are: the natural extension $[f]$, the Taylor extension $[f]_t$, the Hansen extension $[f]_h$, the evaluation by monotonicity $[f]_m$, the recursive evaluation by monotonicity $[f]_{mr}$, the recursive evaluation by monotonicity combined with the Hansen extension $[f]_{mr+h}$ and the recursive evaluation by monotonicity combined with the occurrence grouping extension $[f]_{mr+og}$ (in Section 4 we describe how to combine the recursive extension by monotonicity with other extensions).

It is well-known that the Taylor and Hansen extensions are not comparable with the natural extension. Therefore, to obtain more reasonable comparisons we have re-defines $[f]_t([B]) = [f]'_t([B]) \cap [f]([B])$ and $[f]_h([B]) = [f]'_h([B]) \cap [f]_n([B])$ where $[f]'_t$ and $[f]'_h$ are the real Taylor and Hansen extensions respectively.

The table shows that $[f]_{og}$ computes, in general, sharper evaluations than all the competitors (only $[f]_{mr+og}$ obtains sharper evaluations but also uses occurrence grouping). The improvements w.r.t. the two evaluations by monotonicity methods ($[f]_m$ and $[f]_{mr}$) corroborate the benefits of our approach. For example in `Fourbar`, $[f]_{og}$ obtains an evaluation diameter which is 42.7% of the evaluation diameter provided by $[f]_{mr}$.

$[f]_{mr+h}$ obtains the sharpest evaluations in three benchmarks (`Caprasse`, `Fourbar` and `Virasoro`). How-

[1] See section B.4 for details about the used average method.

ever, $[f]_{mr+h}$ is more expensive than $[f]_{og}$. $[f]_{mr+h}$ requires to compute $2n$ interval partial derivatives traversing $4n$ times the expression tree if the AD method is used (the reason is described in Section 2.4.4). $[f]_{og}$ computes n partial derivatives but traverses the tree only twice.

Cheaper and better than $[f]_{mr+h}$, the extension using occurrence grouping $[f]_{mr+og}$ is *strictly* better in evaluation than $[f]_{og}$. However, the gain in evaluation diameter is only 1.6% on average (between 0% and 6.2%), so that we do not believe it constitutes a promising extension.

5.6.5 Frequency of interesting evaluations

We consider that an evaluation is *interesting* when its diameter is less than 70% (based on the experiments reported in Figure 4.4-right) the diameter of the natural evaluation. Following Table 5.3, Table 5.4 shows the *frequency of interesting evaluations* ($\frac{\#\texttt{interesting\_evaluations}}{\#\texttt{total\_calls}}$) performed by different interval extensions during the solving of each benchmark (using `Mohc`).

NCSP	$[f]_t$	$[f]_h$	$[f]_m$	$[f]_{mr}$	$[f]_{mr+h}$	$[f]_{og}$	$[f]_{mr+og}$
`Brent`	0.09	0.10	0.10	0.10	0.10	0.10	0.10
`ButcherA`	0.5	0.68	0.34	0.56	0.88	0.87	**0.91**
`Caprasse`	0.22	0.29	0.20	0.32	0.39	0.36	**0.41**
`Direct kin.`	0.45	0.54	0.46	0.55	0.64	0.66	**0.68**
`Eco9`	0	0.01	0.01	0.02	0.03	0.03	**0.04**
`Fourbar`	0.78	0.87	0.24	0.31	0.90	0.92	**0.93**
`Geneig`	0.49	0.56	0.47	0.55	0.61	**0.66**	**0.66**
`Hayes`	0.39	0.39	0.39	0.39	0.39	**0.4**	**0.40**
`I5`	0.02	0.02	0.02	0.02	0.03	**0.06**	**0.06**
`Katsura`	0.07	0.10	**0.17**	**0.17**	**0.17**	**0.17**	**0.17**
`Kin1`	0.11	0.11	0.12	0.12	0.13	**0.14**	**0.14**
`Pramanik`	0.38	0.46	0.11	0.12	0.47	**0.51**	**0.51**
`Redeco8`	0	0.03	0.02	0.04	**0.07**	**0.07**	**0.07**
`Trigexp2`	0	0	0	0	0	0	0
`Trigo1`	0.35	0.35	0.34	0.34	0.49	**0.53**	**0.53**
`Virasoro`	0.19	0.24	0.13	0.14	0.28	0.33	**0.34**
`Yamamura1`	0.28	0.28	0.09	0.09	0.29	**0.3**	**0.30**
AVERAGE	0.255	0.296	0.189	0.227	0.345	0.358	**0.368**

Table 5.4: Frequencies of interesting evaluations performed by different interval extensions.

The two extensions based on occurrence grouping report the highest frequencies of interesting evaluations. As in Table 5.3, the results of $[f]_{mr+h}$ are similar to $[f]_{og}$ while obtained at a highest cost.

5.7 Conclusion

We have proposed a new method to improve the monotonicity-based evaluation of a function f. The *Occurrence Grouping* method creates for each variable three auxiliary, respectively increasing, decreasing and non monotonic variables in f. It then transforms f into a function f^{og} that groups the occurrences of a variable into these auxiliary variables. As a result, the *evaluation by occurrence grouping* of f, i.e., the evaluation by monotonicity of f^{og}, is better than the evaluation by monotonicity of f.

Occurrence grouping shows good performances when it is used to improve the monotonicity-based existence test, and when it is embedded in the `Mohc` contractor algorithm that exploits monotonicity of functions.

We still have to compare the occurrence grouping with symbolic-based extensions (see Section 2.4.5). However, monotonic-based and symbolic-based extensions are compatible. For instance, after having reduced the number of occurrences using a Horner-based scheme it is still possible to evaluate the new form using a monotonic-based extension for obtaining a sharper interval.

Chapter 6

Exploiting Common Subexpressions

Contents

Common subexpression elimination (CSE) is a significant feature in compiler optimization [Muchnick, 1997]. CSE searches in the code for common subexpressions with identical evaluation and replaces them by auxiliary variables.

This technique has also been applied to interval analysis for reducing the number of operations in the function evaluations [Granvilliers et al., 2001; Schichl and Neumaier, 2005; Vu et al., 2004, 2009b]. For more details please refer to Section 3.4.1.

In this chapter we show that the benefits of CSE in interval analysis are even greater. We clearly state which CSs are useful for bringing *a better contraction/filtering*. This is the topic of Section 6.1. Section 6.2 presents a new algorithm `I-CSE` (Interval CSE) to detect CSs and to generate a new system of constraints. As opposed to existing algorithms, and for a given form of equations, `I-CSE` is able to find *all* the "useful" CSs shared by every pair of expressions. This is mainly due to the fact that we find all the n-ary maximal CSs corresponding to sums and products, modulo the commutativity and the associativity of these operators, including so-called *conflictive* CSs that overlap. Finally, experiments shown in Section 6.4 highlight that, in all the tested systems, the CSs are extracted in less than one second. The transformed system of equations then leads standard interval-based solving strategies using `HC4` to significant gains in performance (of sometimes several orders of magnitude).

(A part of the material presented in this chapter is published in [Araya et al., 2008a,b].)

6.1 Properties of HC4 and CSE

We call *Common Subexpression* (in short CS) a numerical expression that occurs several times in one or several constraints. This section investigates when it is useful to extract a CS f from a system, create a new *auxiliary variable* v and add the equation $v = f$ into the system.

If we observe carefully the `HC4-revise` algorithm, we can note that the contraction obtained by a narrowing operator on a given expression f is in general partially lost in the next evaluation of f. Consider for instance a sum $x+z$ that is shared by two expressions n_1 and n_2. Following Figure 6.1, the narrowing phase of `HC4-revise` applied to n_1 contracts its interval to $[-2,5]$. Then, when the evaluation phase of `HC4-revise` applies to n_2, its interval is set to $[-2,6]$ (Figure 6.1–left). Clearly, the interval of n_2 is larger than that of n_1. To avoid this loss of information, the idea is to replace n_1 and n_2 by a common variable v, and to add a new constraint $v = x + z$. The new system is equivalent to the original one (both have the same solutions) while it improves the contraction power of HC4. The introduction of v (Figure 6.1–right) amounts to adding a redundant equation $n_1 = n_2$ (Figure 6.1–center).

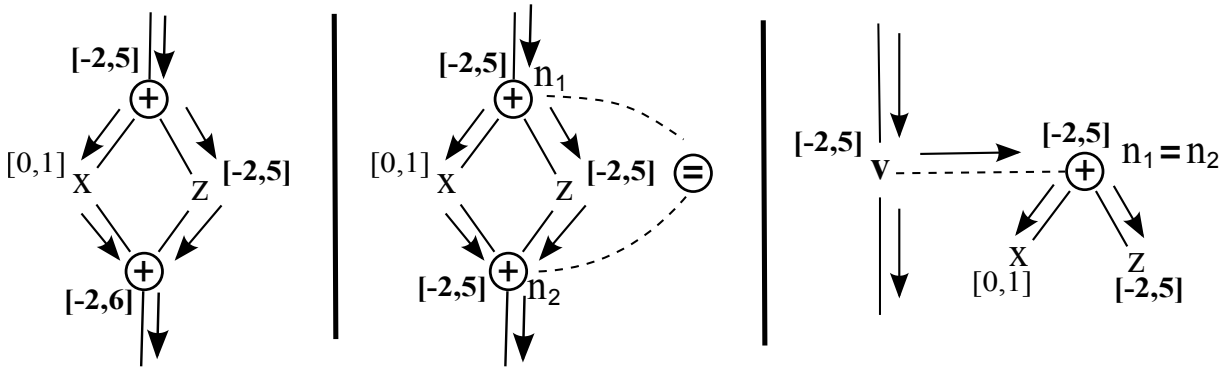

Figure 6.1: Narrowing/Evaluation without and with CSE.

6.1.1 Additional propagation

Proposition 21 underlines that `HC4` obtains a better contraction/filtering when we add into a system new auxiliary variables and equations corresponding to CSs.

Proposition 21 *Let S be a system of equations and S' be the system obtained by replacing in S one CS f in common between two expressions (belonging to constraints in S) by an auxiliary variable v, and by adding the new equation $v = f$.*

Then, `HC4` *(with a floating-point precision) applied to S' produces a contracted box $[B']$ that is smaller than or equal to the box $[B]$ produced by* `HC4` *applied to S.*

Proof 9 *One first produces a system S_1 by replacing in S the first occurrence of f by an auxiliary variable v_1 and the second occurrence of f by an auxiliary variable v_2 and by adding equations $v_1 = f$ and $v_2 = f$.* `HC4` *works with a decomposed system (i.e., ternary system equivalent to S where all the operators are replaced by auxiliary variables, see Section 3.2.3.4). It appears that S and S_1 lead to the same ternary system S_2 (v_1 and v_2 provide a subset of the auxiliary variables). In other words,* `HC4` *(with a floating-point precision) applied to S, S_1 or S_2 produce the same contracted box $[B]$ [Collavizza et al., 1999]. Finally, creating S' amounts in adding the constraint $v_1 = v_2$ to S_1. Thus, the box $[B']$ is smaller than or equal to the box $[B]$.* □

Of course, this result would be useless if the box $[B']$ was equal to $[B]$, and we want to determine conditions for obtaining a box $[B']$ that might be *strictly* smaller than $[B]$. Among the set of *basic operators* that are defined in a standard implementation of `HC4` (the basic operators are described in

Section 2.2), the analysis presented below highlights that the following subset of non-monotonic or non-continuous operators might bring additional contraction when they occur several times (as CS) in the same system: $sin(x)$, $cos(x)$, $tan(x)$ with non-monotonic domains, x^{2c} (c positive integer and $0 \in [x]$), $cosh(x)$ with $0 \in [x]$, $1/x$ with $0 \in X$ and binary operators $(+,-,\times,/)$.

6.1.2 Unary operators

Recall that the interval extension of a function f computes a *conservative* interval containing the image of a given domain $[B]$ under f (see Section 2.4). In other words, the application of f to any element of $[B]$ falls inside the computed interval.

A unary interval operator $[f]([x])$ computes the smallest interval containing the image of $[x]$ under f. Unary operators are defined in Section 2.2.2:

$$[f]([x]) := \texttt{Hull}(\mathcal{I}f([x])) = [\min_{x\in[x]} f(x), \max_{x\in[x]} f(x)] \tag{6.1}$$

A *narrowing* operator $N_x^{f(x)}$ allows us to filter/contract the domain of a variable x using the constraint $c : w = f(x)$ with domains $[w]$ and $[x]$. Narrowing operators are defined in Section 3.2.3.4:

$$N_x^{f(x)}([w],[x]) = \texttt{Hull}(\Pi_x^c([w],[x]))$$

where Π_x^c is the *projection* of c over x (the projection of a constraint is defined in Section 3.2.3.2). The narrowing removes *all* the values from the bounds of $[x]$ not satisfying the constraint $w = f(x)$. i.e., a narrowing operator $[x'] = N_x^{f(x)}([w],[x])$ verifies:

$$f(\underline{x'}), f(\overline{x'}) \in [w] \quad \text{and} \quad (\forall w \in [w]) : \; x \in [x] \wedge x \notin [x'] \Rightarrow f(x) \neq w \tag{6.2}$$

A necessary condition to replace a CS f while bringing additional propagation is when the contraction obtained by the narrowing operator on f is partially lost in the next evaluation of f (recall the example of Figure 6.1). More formally:

Condition 1

$$\exists [w] \subseteq [f]([x]), [x'] = N_x^{f(x)}([w],[x]) \textit{ such that } [f]([x']) \not\subseteq [w]$$

where $[f]$ *is the interval operator related to* f*.*

The following proposition indicates a simple condition to identify a *useless CS* for which no filtering is expected.

Proposition 22 *Let* $[f]$ *be the unary interval operator associated with the unary operator* f*. Let* $N_x^{f(x)}$ *be the narrowing operator of* f *over the variable* x*.* $[x]$ *is the interval related to* x*. If* f *is continuous and monotonic w.r.t.* x[1]*, then:*

$$\forall [w] \subseteq [f]([x]), [x'] = N_x^{f(x)}([w],[x]) : \quad [f]([x']) \subseteq [w]$$

[1] A continuous and differentiable function is monotonic w.r.t. a variable x if the derivative is positive (or negative) in all the domain of x. See Section 2.4.2.

Proof 10 *W.l.o.g. we suppose that f is monotonically increasing. $[x'] = N_x^{f(x)}([w],[x])$, then using (6.2): $f(\underline{x'}), f(\overline{x'}) \in [w]$. Then, using (6.1) and the fact that f is monotonic:*

$$[f]([x']) = [\min_{x\in[x]} f(x), \max_{x\in[x]} f(x)] = [f(\underline{x'}), f(\overline{x'})] \subseteq [w] \quad \square$$

Proposition 23 *With the same notations as Proposition 22, if f is a non-monotonic function, then:*

$$\exists [w] \subseteq [f]([x]), [x'] = N_x^{f(x)}([w],[y]) : f([x']) \not\subseteq [w]$$

Proof 11 *The non monotonicity of f means:*

$$\exists x_1, x_2, x_3 \subseteq [x]^3,\ x_1 \leq x_2 \leq x_3 \text{ s.t. } f(x_2) > f(x_1) \ \wedge\ f(x_2) > f(x_3)$$

Using values x_1, x_2 and x_3 that satisfy the existence condition, we can suppose that $[w] = [f(x_1), f(x_3)]$. As $(f(x_2) > f(x_1)) \wedge (f(x_2) > f(x_3))$, $f(x_2) \notin [w]$. $[x'] = N_x^{f(x)}([w],[x])$, then, with (6.2), $[x_1, x_3] \subseteq [x']$. Since $x_1 \leq x_2 \leq x_3$, $x_2 \in [x']$. With (6.1), $f(x_2) \in [f]([x'])$ implying $[f]([x']) \not\subseteq [w]$. $\square$

Example 28 *Let $[x] = [-1, 3]$ be the domain of a variable x, and x^2 be an expression shared by two or more constraints. Suppose that in the narrowing phase of* `HC4-revise`*, the node corresponding to one of the expressions, i.e., $w = x^2$, is contracted to: $[w] = [3, 4]$. Applying the narrowing operator on x contracts the interval to $[x] \leftarrow N_x^{x^2}([3,4],[-1,3]) = [-1, 2]$. In the next evaluation of the expression, $[f]([x]) = [0, 4] \not\subseteq [w]$.*

Considering the standard operators managed in `HC4` (except operators like `floor`), the *useful CSs* do not satisfy Proposition 22 and satisfy Proposition 23 (e.g., x^2, *sin*, *cos*).

6.1.3 N-ary operators (sums, products)

For binary (n-ary) primitive functions, Condition 1 can be extended to the following condition:

Condition 2

$$\exists [w] \subseteq [f]([x],[y]), [x'] = N_x^{f(x,y)}([w],[x],[y]), [y'] = N_x^{f(x,y)}([w],[x],[y]) \text{ s.t. } [f]([x'],[y']) \not\subseteq [w]$$

where $[f]$ is the binary operator associated to f, $N_x^{f(x,y)}$ and $N_y^{f(x,y)}$ are the narrowing operators of the constraint $c : w = f(x, y)$ over x and y resp. (binary narrowing operators are described in Section 3.2.3.4).

Condition 2 is generally satisfied by the n-ary operators $+$ and $\times$ (resp. $-$ and $/$). Many examples prove this result (see Example 29). The result is due to intrinsic "bad" properties of interval arithmetic. First, the set of intervals $\mathbb{IR}$ is not a group for addition. That is, let $[x]$ be an interval: $[x] - [x] \neq [0, 0]$ (in fact, $[0, 0] \subset [x] - [x]$). Second, $\mathbb{IR} \setminus \{0\}$ is not a group for multiplication, i.e., $[x]/[x] \neq [1, 1]$.

Proposition 24 provides a *quantitative idea* of *how much* we can win when replacing additive CSs. It estimates the width Δ that is lost in binary sums when an additive CS is not replaced by an auxiliary variable. Note that an upper bound of Δ is $2 \times \min(\texttt{Diam}([x]), \texttt{Diam}([y]))$ and depends only on the initial domains of the variables. The lower bound is always positive (or 0) and depends on the reduction of $[w]$ and on the diameters of the initial variables.

Proposition 24 *Let $x + y$ be a sum related to a node $w = x + y$ inside the tree representation of a constraint. The domains of x and y are the intervals $[x]$ and $[y]$ resp. Suppose that* `HC4-revise` *is carried out on the constraint: in the evaluation phase, the interval of the node is set to $[w] = [x] + [y]$; in the narrowing phase, the interval $[w]$ is contracted to $[w'] = [\underline{w} + \alpha, \overline{w} - \beta]$ (with $\alpha, \beta \geq 0$ being the reduction in left and right bounds of $[w]$); $[x]$ and $[y]$ are contracted to $[x']$ and $[y']$ resp. The difference Δ between the diameter of the sum $[x'] + [y']$ (computed in the next evaluation) and the diameter of $[w']$ (current projection) is:*

$$\Delta = \min(\alpha, Diam([x]), Diam([y]), Diam([w]) - \alpha) \; + \; \min(\beta, Diam([x]), Diam([y]), Diam([w]) - \beta) \qquad (6.3)$$

Proof 12 *The proof of Proposition 24 can be obtained knowing that $[x'] = N_x^{x+y}([w'], [x], [y]) = [x] \cap ([w] - [y])$ (see how the narrowing operators are obtained in Section 3.2.3.4) and $[y'] = N_y^{x+y}([w'], [x], [y]) = [y] \cap ([w] - [x])$. If $[w']$ is replaced by $[\underline{x+y} + \alpha, \overline{x+y} - \beta]$ then we obtain (with few calculations):*

$$\begin{array}{rcl} [x'] & = & [\underline{x} + \max(0, \alpha - \texttt{Diam}([y])), \overline{x} - \max(0, \beta - \texttt{Diam}([y]))] \\ [y'] & = & [\underline{y} + \max(0, \alpha - \texttt{Diam}([x])), \overline{y} - \max(0, \beta - \texttt{Diam}([x]))] \end{array}$$

Finally $\Delta = \texttt{Diam}([x'] + [y']) - \texttt{Diam}([w'])$ is given by (6.3). □

Example 29 *Consider $[x] = [0, 1]$ and $[y] = [2, 4]$. Thus, $[w] = [x] + [y] = [2, 5]$. Suppose that after applying* `HC4-Revise` *we obtain $[w'] = [2 + \alpha, 5 - \beta] = [4, 4]$ ($\alpha = 2$, $\beta = 1$). With Proposition 24 we obtain $\Delta = 2$. The narrowing operator yields $[x'] = [0, 1]$ and $[y'] = [3, 4]$. Indeed, $[x'] + [y'] = [3, 5]$ is $\Delta = 2$ units larger than $[w'] = [4, 4]$.*

The properties related to multiplication are more difficult to establish and understand. Concise results have been obtained only in the cases when 0 does not belong to the domains or when 0 is a bound of the domains. First, let us define the **ratio** of an interval:

$$\texttt{Ratio}([x]) = \frac{\max_{x \in [x]}(|x|)}{\min_{x \in [x]}(|x|)} = \frac{|[x]|}{\langle [x] \rangle}$$

Proposition 25 *Let $x\,y$ be a product related to a node $w = xy$ inside the tree representation of a constraint. The domains of x and y are the intervals $[x] \geq 0$ and $[y] \geq 0$ resp. Suppose that* `HC4-revise` *is carried out on the constraint: in the evaluation phase, the interval of the node is set to $[w] = [x] \times [y]$; in the narrowing phase, the interval $[w]$ is contracted to $[w'] = [\underline{w} \times \alpha, \overline{w}/\beta] \geq 0$ (with $\alpha, \beta \geq 1$); $[x]$ and $[y]$ are contracted to $[x']$ and $[y']$ resp. The quotient $\Theta = \frac{\texttt{Ratio}([x'] \times [y'])}{\texttt{Ratio}([w'])}$ between the ratio of the product $[x'] \times [y']$ (computed in the next evaluation) and the ratio of $[w']$ (current projection) is:*

$$\Theta = \min\left(\alpha, \texttt{Ratio}([x]), \texttt{Ratio}([y]), \frac{|[w]|}{\langle [w'] \rangle}\right) \; \times \; \min\left(\beta, \texttt{Ratio}([x]), \texttt{Ratio}([y]), \frac{|[w']|}{\langle [w] \rangle}\right) \qquad (6.4)$$

Proof 13 *The proof is analogously obtained from Proposition 24 by replacing the occurrences of* `Diam` *by* `Ratio`*, sums by products, subtractions by divisions and 0s by 1s.*

From Proposition 25 we deduce $1 \leq \Theta \leq (min\,(\texttt{Ratio}([x]), \texttt{Ratio}([y])))^2$. Observe the similarities with Proposition 24 (using `Ratio` instead of `Diam`).

Example 30 *Consider* $[x] = [1,3]$ *and* $[y] = [4,8]$. *Thus,* $[w] = [x] \times [y] = [4,24]$. *Suppose that after applying* `HC4-Revise` *we obtain* $[w'] = [4\alpha, \frac{24}{\beta}] = [4,6]$ *(*$\alpha = 1$, $\beta = 4$*). Using (6.4) we compute* $\Theta = \alpha \times \frac{|[w']|}{\langle [w] \rangle} = \frac{3}{2}$. *The narrowing operator yields* $[x'] = [1, \frac{3}{2}]$ *and* $[y'] = [4,6]$. *We can check that the quotient between* $\texttt{Ratio}([w']) = \frac{3}{2}$ *and* $\texttt{Ratio}([x'] \times [y']) = \texttt{Ratio}([4,9]) = \frac{9}{4}$ *is* $\frac{\frac{9}{4}}{\frac{3}{2}} = \frac{3}{2}$.

6.2 The `I-CSE` algorithm

Like classical CSE techniques, our algorithm `I-CSE` detects CSs in a system of constraints and replaces them by auxiliary variables, which generates a new system.

The novelty of our `I-CSE` lies in the way additive and multiplicative CSs are taken into account.

First, `I-CSE` manages the commutativity and associativity of $+$ and $\times$ in a simple way thanks to *intersections* between expressions.

Definition 19 *(***Intersection between two sums**[1]*) Consider the sum expressions* $f = a_1 + ... + a_n$ *and* $f_2 = b_1 + ... + b_m$. *W.l.o.g. the* **terms** *of* f_1 *and* f_2 *(i.e.,* $a_1, ..., a_k$ *and* $b_i, ..., b_m$ *resp.) are arithmetic expressions different from a sum (e.g.,* x, xy, $sin(x+z)$, $(x+3)(z*y)$*).*

If $\{c_1, ..., c_k\} = \{c, c \in \{a_1, ..., a_n\}, c \in \{b_1, ..., b_m\}\}$ *then the intersection* $(\cap)$ *between* f_1 *and* f_2 *is defined:*

$$f_1 \cap f_2 = c_1 + ... + c_k$$

Example 31 *The intersection between the expressions* $f_1 = x + y \times (z + x^2) + 5z$ *and* $f_2 = x^2 + x + 5z$ *is:*

$$f_1 \cap f_2 = x + 5z$$

Consider two expressions $w_1 \times x \times y \times z_1$ *and* $w_2 \times y \times x \times z_2$ *that share the CS* $x \times y$. *We are able to view these two expressions as* $w_1 \times (x \times y) \times z_1$ *and* $w_2 \times (x \times y) \times z_2$ *since* $(w_1 \times x \times y \times z_1) \cap (w_2 \times y \times x \times z_2) = x \times y$.

Second, contrarily to existing CSE algorithms, `I-CSE` handles *conflictive* subexpressions.

Definition 20 *(***Conflictive subexpression***) Two CSs* f_a *and* f_b *included in* f *are* **in conflict** *(or* **conflictive***) if* $f_a \cap f_b \neq \emptyset$, $f_a \not\subseteq f_b$ *and* $f_b \not\subseteq f_a$.

An example of conflictive CSs occurs in the expression $f : x \times y \times z$ that contains the conflictive CSs $f_a : x \times y$ and $f_b : y \times z$. Since $x \times y$ and $y \times z$ have a non empty intersection y, it is not possible to directly replace both f_a and f_b in f.

`I-CSE` works with n-ary trees encoding the original equations[2] and produces a DAG. The roots of this DAG correspond to the initial equations; the leaves correspond to the variables and constants; every internal node f corresponds to an *operator* $(+, \times, sin, exp,$ etc.) applied to its *children* $t_1, t_2 ..., t_n$. f represents the expression $f(t_1, t_2, ..., t_n)$. *The CSs are represented by nodes with several parents.*

[1] The definition is straightforwardly extended to products.

[2] The $+$ and $\times$ operators are viewed as **n-ary** operators. They include $-$ and $/$. For example, the 3-ary expression $x^2 y/(2+x)$ is viewed as $\times(x^2, y, \frac{1}{(2+x)})$.

Example 32 *We illustrate* I-CSE *with the following system made of two equations.*

$$\begin{aligned} x^2 + y + (y + x^2 + y^3 - 1)^3 + x^3 &= 2 \\ \frac{(x^2 + y^3)(x^2 + cos(y)) + 14}{x^2 + cos(y)} &= 8 \end{aligned}$$

Algorithm 14 I-CSE(**in:** $eqSys$; **out:** $newSys$)

```
1: /* Step 1: */
2: nodeList ← DagGeneration(eqSys) /* The list must be in increasing order */
3: /* Step 2: */
4: PairwiseIntersection(GetNodes(Plus), nodeList)
5: PairwiseIntersection(GetNodes(Times), nodeList)
6: /* Step 3: */
7: for all node ∈ nodeList such that node.CS_list ≠ ∅ do
8:    ProcessNode(node)
9: end for
10: /* Step 4: */
11: newSys ← SystemGeneration(nodeList)
```

6.2.1 Step 1: DAG generation

This step corresponds to the DagGeneration *procedure of the* I-CSE *algorithm.*

This step follows a standard CSE algorithm that traverses simultaneously the n-ary trees corresponding to the equations in a bottom-up way (see e.g., [Flajolet et al., 1990]). By labeling nodes with identifiers, two nodes with common children and with the same operator are identified equivalent, i.e., they are CSs. Figure 6.2 shows the DAG obtained by this algorithms applied to Example 32. The nodes 11, 12 and 13 correspond to CSs. (For the sake of clarity, we have not merged in a same node the multiple occurrences of a same variable.)

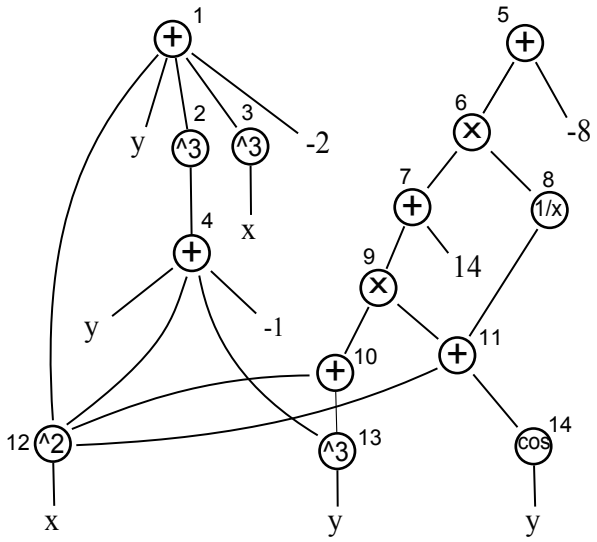

Figure 6.2: DAG obtained after the first step of I-CSE in Example 32.

At the end, the procedure returns the list *nodeList* of nodes in this DAG. The vertices of the DAG are included in a node structure (e.g., $n.children$ corresponds to the list of children of the node n).

6.2.2 Step 2: Pairwise intersection between sums and products

This step corresponds to the `PairwiseIntersection` *procedure called twice by the* `I-CSE` *algorithm.*

Step 2 pairwise intersects, in any order, the nodes corresponding to n-ary sums on one hand, and to n-ary products on the other hand (see Algorithm 15). Each intersection extracts the maximal CS (if any) shared by a pair of nodes (line 3). If the CS is useful then it is added in a CS list maintained in each node (CS_{list}, lines 5-6). The CS list of each node is initialized as an empty set when the node is created. Note that the number of terms (`#Children`) in the CS must be at least 2 to be considered useful (line 4).

Algorithm 15 `PairwiseIntersection`(**in-out:** *nodes*, *nodeList*)

```
1: for i = 1 to length(nodes) do
2:   for j ← i + 1 to length(nodes) do
3:     csnode ← Intersection(nodes[i], nodes[j], nodeList)
4:     if #Children(csnode) ≥ 2    /* useful CS */ then
5:       if csnode ≠ nodes[i] then nodes[i].CS_list.add(csnode)  end if
6:       if csnode ≠ nodes[j] then nodes[j].CS_list.add(csnode)  end if
7:     end if
8:   end for
9:   /* The list nodes[i].CS_list must be sorted by decreasing expression sizes. */
10:  for all csnode_1 ∈ nodes[i].CS_list do
11:    UpdateCSLists(csnode_1, nodes[i].CS_list)
12:  end for
13: end for
```

Algorithm 16 `UpdateCSLists`(**in-out:** cs_1, CS_{list})

```
1: for all cs_2 ∈ CS_list such that cs_2.expr ⊂ cs_1.expr do
2:   CS_list.remove(cs_2)
3:   cs_1.CS_list.add(cs_2) /* only if cs_2 ∉ cs_1.CS_list */
4:   UpdateCSLists(csnode_2, CS_list)
5: end for
```

The `Intersection` procedure intersects the expressions corresponding to two nodes i and j. If the intersection expression corresponds to a node included in *nodeList*, then the procedure returns this node. Otherwise a new node corresponding to the intersection expression is added to *nodeList* and returned.

On Example 32 (see Figure 6.3), the intersection between the node 1 and the node 4 generates a new node:

$$node_{1.4} : y + x^2 = \texttt{Intersection}(node_1, node_4).$$

The intersection between the node 4 and node 10 corresponds to the expression: $x^2 + y^3$. In this case the expression is related to an existing node ($node_{10}$) so that no new node is created.

Also, the list of CSs is updated for each node in the DAG:

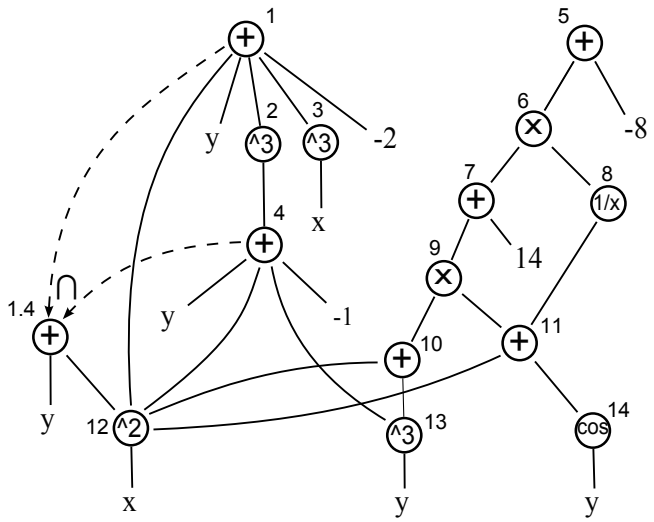

Figure 6.3: DAG obtained after the second step of `I-CSE` in Example 32: The intersection between nodes 1 and 4 creates the node 1.4.

$$node_1.CS_{list} = \{node_{1.4}\}$$
$$node_4.CS_{list} = \{node_{1.4}, node_{10}\}$$

The `for` loop (lines 10-12) and the `UpdateCSLists` procedure update the CS lists of each node and of its CSs (recall that a CS is also a node). `UpdateCSLists` applies the following treatment to each group of three nodes. Consider the nodes g, cs_1 and cs_2, with CS lists: $g.CS_{list} = \{cs_1, cs_2, ...\}$, $cs_1.CS_{list} = \{...\}$ and $cs_2.CS_{list} = \{...\}$. If the expression related to cs_2 is contained by the expression related to cs_1 (i.e., $cs_2.expr \subset cs_1.expr$), then the procedure removes cs_2 from $g.CS_{list}$ and adds it to $cs_1.CS_{list}$:

$$\begin{array}{rcl} g.CS_{list} & = & \{cs_1, cs_2, ...\} \\ cs_1.CS_{list} & = & \{...\} \\ cs_2.CS_{list} & = & \{...\} \end{array} \quad \rightarrow \quad \begin{array}{rcl} g.CS_{list} & = & \{cs_1, ...\} \\ cs_1.CS_{list} & = & \{cs_2, ...\} \\ cs_2.CS_{list} & = & \{...\} \end{array}$$

As a result, only the maximal CSs are kept in the CS lists of each node.

The following example describes line by line Algorithm 16.

Example 33 *Consider a node $node[i] = f$ corresponding to the expression $x_1 + x_2 + x_3 + x_4 + x_5$, that contains, initially, four CSs: $f_a : x_1 + x_2 + x_3 + x_4$, $f_b : x_1 + x_2$, $f_c : x_3 + x_4$ and $f_d : x_4 + x_5$ (i.e., $f.CS_{list} = \{f_a, f_b, f_c, f_d\}$). Figure 6.4-top shows the different nodes/expressions.*

The first time the `UpdateCSLists` *is called, it is called with $cs_1 = f_a$ and $CS_{list} = \{f_a, f_b, f_c, f_d\}$. The procedure detects that $cs_2 = f_b \subset f_a$ (i.e. $(x_1 + x_2) \subset (x_1 + x_2 + x_3 + x_4)$), then the corresponding CS lists are updated as follows: f_b is removed from the CS list of f (second line of Algorithm 16) and added into the CS list of f_a (third line of the algorithm), i.e.:*

$$f.CS_{list} = \{f_a, f_c, f_d\}$$
$$f_a.CS_{list} = \{f_b\}$$

Then a recursive call to `UpdateCSLists` *is performed with $cs_1 = f_b$ and $CS_{list} = \{f_a, f_c, f_d\}$. As there does not exist any CS in $f.CS_{list}$ included in f_b, the recursion terminates. Observe that f_b has been removed from $f.CS_{list}$ and is not handled again by the* `for` *loop of Algorithm 15 (CS_{list} is an* `in-out` *parameter*

of the procedure `UpdateCSLists`*). This explains why the CS list of* f *must be sorted by decreasing sizes of expressions.*

The algorithm continues handling the node $cs_1 = f_a$ *and detects that* $cs_2 = f_c \subset f_a$*. Thus, the final CS lists are (nothing happens when the nodes* f_c *and* f_d *are handled by* `UpdateCSLists`*):*

$$f.CS_{list} = \{f_a, f_d\}$$
$$f_a.CS_{list} = \{f_b, f_c\}$$

Step 2 is a key step because it makes appear CSs modulo the commutativity and associativity of + and $\times$ operators, and creates at most a quadratic number of CSs.

6.2.3 Step 3: Integrating CSs into the DAG

This step corresponds to the `for` *loop of the* `I-CSE` *algorithm.*

In step 3, all the CSs are integrated into the DAG, thus creating the final DAG. The routine can treat the nodes of *nodeList* in any order. Each node (containing CSs) is processed by Algorithm 17 to incorporate the CSs as children.

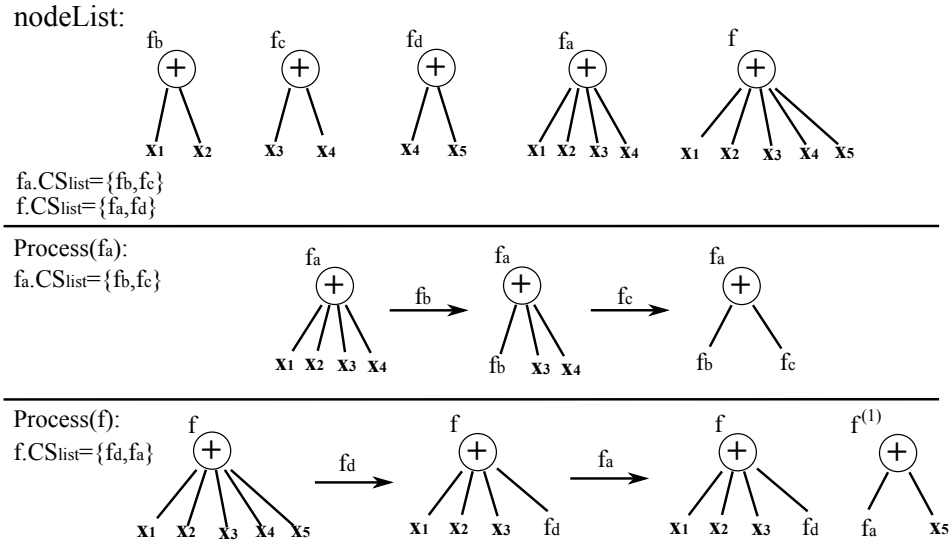

Figure 6.4: Illustration of the `Process` procedure in Example 33.

Let us explain Algorithm 17 on using Example 33 and Figure 6.4. The *nodeList* is composed of 5 nodes, two of them containing CSs (f and f_a). When $node = f_a$ is processed, a temporary copy *tmpnode* is first created in line 3. Then, $remaining[tmpnode]$ saves the *remaining* expression of the node $(x_1+x_2+x_3+x_4)$ i.e., the subpart of the expression f that has not been replaced yet by a CS. The *remaining* array is in fact a hash table able to recover the remaining expression of a node n with its identifier n (i.e., $remaining[n]$).

The for loop (lines 5-11) adds the nodes in $f_a.CS_{list}$ as children of *tmpnode* if they are not conflictive (see Figure 6.4-middle). For instance, when f_b is handled by the loop, line 6 checks if the node $csnode = f_b$ can replace the expression $csnode.expr = x_1 + x_2$ in the node *tmpnode*, i.e., if $csnode.expr \subseteq remaining[tmpnode]$. As the condition is satisfied, f_b is added as a child of *tmpnode* (line 7), the remaining expression is updated to $x_3 + x_4$ (line 8) and f_b is removed from the CS list (line 9). In the same way, f_c is added as a child of *tmpnode*, replacing leaves x_3 and x_4 (see the figure). Then,

the for and repeat-until loops terminate because CS_{list} is empty. Finally, $node = f_a$ obtains its children from the temporary node (line 14).

Algorithm 17 `Process`(**in:** *node*)

```
1: CS_list ← node.CS_list;  redundNodes ← {};  node_0 ← null
2: repeat
3:    tmpnode ← newNode(node.expr)
4:    remaining[tmpnode] ← node.expr
5:    for all csnode ∈ CS_list do
6:       if csnode.expr ⊆ remaining[tmpnode] then
7:          tmpnode.addChild(csnode)
8:          remaining[tmpnode] ← Complement(remaining[tmpnode], csnode.expr)
9:          CS_list.remove(csnode)
10:      end if
11:   end for
12:   if node_0 = null then node_0 ← tmpnode else redundNodes.add(tmpnode) end if
13: until CS_list = {}
14: node.children ← node_0.children
15: node.redundNodes ← redundNodes
16: [Simplify(node, redundNodes, remaining) /* (see Section 6.2.5) */]
```

When $node = f$ is processed (Figure 6.4-bottom), after adding f_d in the for loop, the remaining expression is updated to $remaining[tmpnode] = x_1 + x_2 + x_3$. Then, $csnode = f_a$ cannot be added to f because it is in conflict with f_d ($f_a \cap f_d = x_4 \neq \emptyset$). The algorithm detects the conflict in line 6: $csnode.expr \not\subseteq remaining[tmpnode]$ (i.e., $x_1 + x_2 + x_3 + x_4 \not\subseteq x_1 + x_2 + x_3$).

In case of conflict, the repeat-until loop iterates. In each iteration a new redundant node *tmpnode* is created (line 3). Redundant nodes are maintained in the list *redundNodes*. The loop terminates when each CS in the CS list has been added as child of a redundant node (line 7). At the end of the process (line 15) the list of redundant nodes (*redundNodes*) is associated to the current node *node*.

Following with the example, as the node f_a is not added as child of *tmpnode* in the first iteration, a redundant node is created. The process is repeated and f_a is finally added as child of the redundant node ($f^{(1)}$ in the figure).

As a result, the procedure has symbolically transformed the expression $f = x_1 + x_2 + x_3 + x_4 + x_5$, by adding the CSs f_a and f_d, into two redundant expressions:

$$\begin{aligned} f &= x_1 + x_2 + x_3 + f_d \\ f^{(1)} &= f_a + x_5 \end{aligned}$$

In the same way, for the Example 32 (see Figure 6.5), when the node 1 is handled by the algorithm, the node 1.4 is added as a child of it. Also, when the node 4 is handled, a redundant node is created (4[1]). The node 1.4 is then added as a child of node 4 and the node 10 is added as a child of node 4[1]. In the figure we use a dashed line to indicate that the redundant node is associated to the node 4.

Remark: In case of conflicts, Algorithm 17 can generate different DAGs depending on the order of the nodes in the CS lists. Otherwise, if there are not conflicts, the generated DAG is unique.

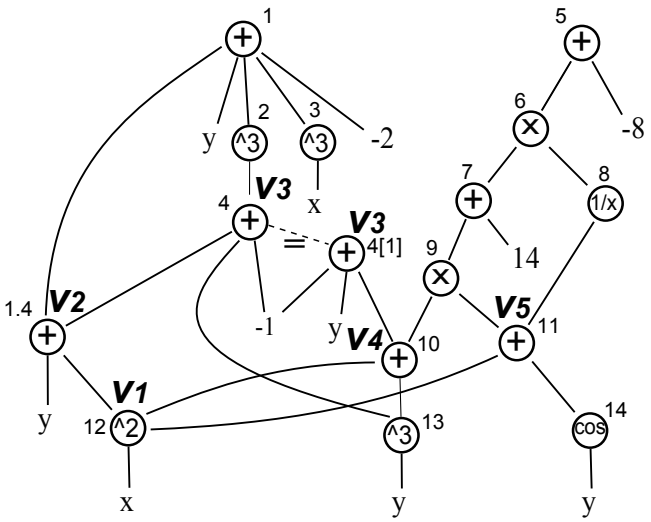

Figure 6.5: DAG obtained after the third and fourth steps of `I-CSE` in Example 32. **Step 3:** All the CSs have been added as children of the nodes. For the conflictive subexpressions (nodes 1.4 and 10), a redundant node 4[1] has been created. The node 1.4 is attached to 4 whereas the node 10 is attached to 4[1]. **Step 4** generates the auxiliary variables corresponding to the useful CSs (v_1, v_2, v_4 and v_5) and the conflictive nodes (v_3).

6.2.4 Step 4: Generation of the new system

This step corresponds to the `SystemGeneration` *procedure of the* `I-CSE` *algorithm.*

A first way to exploit CSs for solving an NCSP is to use the DAG obtained after Step 3. As shown by Vu et al. in [Vu et al., 2004, 2009b], the propagation phase cannot still be carried out by a pure `HC4` algorithm, and a more sophisticated propagation algorithm must consider the unique DAG corresponding to the whole system.

Alternatively, in order to still be able to use `HC4` for propagation, and thus to be compatible with existing interval-based solvers, Step 4 generates a new system of equations in which an auxiliary variable v and an equation $v = f$ are added for every *useful* CS. Avoiding the creation of new equations for useless CSs, which cannot provide additional contraction, decreases the size of the new system. In addition, an auxiliary variable v' and the equations $v' = f$, $v' = f^{(1)}$,...,$v' = f^{(r)}$ are added for every node having redundancies (i.e., conflictive CS). To achieve these tasks, Step 4 traverses the DAG bottom-up and generates variables and equations in every node. In Figure 6.5, the nodes 12, 1.4, 10 and 11 correspond to useful CSs. They generate the auxiliary variables v_1, v_2, v_4 and v_5, and the equations $v_1 = x^2$, $v_2 = y + v_1$, $v_4 = v_1 + y^3$ and $v_5 = v_1 + cos(y)$ respectively. The redundant nodes 4 and 4[1] generate the auxiliary variable v_3 and the redundant equations $v_3 = v_2 + y^3 - 1$ and $v_3 = -1 + y + v_4$. The root nodes (nodes 1 and 5) generate the two modified equations from the original system: $v_2 + (v_3)^3 + x^3 - 2 = 0$ and $\frac{v_4 \times v_5 + 14}{v_5} - 8 = 0$.

Finally, the new system is given by:

$$\begin{array}{rcl} v_2 + (v_3)^3 + x^3 - 2 & = & 0 \\ \dfrac{v_4 \times v_5 + 14}{v_5} - 8 & = & 0 \end{array} \qquad \begin{array}{rcl} v_1 & = & x^2 \\ v_2 & = & y + v_1 \\ v_3 & = & v_2 + y^3 - 1 \end{array} \qquad \begin{array}{rcl} v_3 & = & -1 + y + v_4 \\ v_4 & = & v_1 + y^3 \\ v_5 & = & v_1 + cos(y) \end{array}$$

For a given system of equations, our interval-based solver manages two systems: the new system generated by I-CSE is used only for HC4 and the original system is used for the other operations (bisections, interval Newton). The intervals in both systems must be synchronized during the search of solutions. First, this allows us to clearly validate the benefits of I-CSE for HC4. Second, carrying out Newton or bisection steps on auxiliary variables would need to be validated both in theory and in practice. Finally, this implementation is similar to the DAG-based solving algorithm proposed by Vu et al. which also considers only the initial variables for bisections and interval Newton computations, the internal nodes corresponding to CSs being only used for propagation [Vu et al., 2004, 2009b].

6.2.5 Advanced Feature: Simplification of redundant expressions

For a node f, when the Process procedure creates one or more redundant nodes, Algorithm 18 is performed for reducing the number of children in the redundant nodes and improving filtering. Consider the expression/node $f : s + t + x + y + z$ containing 3 CSs: $cs_1 : s + t$, $cs_2 : t + x$, $cs_3 : y + z$ (i.e., $f.CS_{list} = \{cs_1, cs_2, cs_3\}$). When Process handles the node f, the following expressions are obtained:

$$\begin{aligned} f &: cs_1 + x + cs_3 \\ f^{(1)} &: s + cs_2 + y + z \end{aligned}$$

where $f^{(1)}$ is a redundant node.

Algorithm 18 Simplify(**in:** *node*, *redundNodes*, *remaining*)

```
1: for all rnode ∈ redundNodes do
2:   for all csnode ∈ node.CS_list do
3:     if csnode.expr ⊆ remaining[rnode] then
4:       rnode.addChild(csnode)
5:       remaining[rnode] ← Complement(remaining[rnode], csnode.expr)
6:     end if
7:   end for
8: end for
```

The Simplify procedure handles each redundant node *rnode* created by Process. Following the example, consider the node $rnode = f^{(1)}$. If the condition of line 3 is satisfied, then the for loop of lines 2-7 adds, in a geedy way, CSs in $f.CS_{list}$ as children of $f^{(1)}$. As a result, only cs_3 is added in our example:

$$f^{(1)} : s + cs_2 + cs_3$$

6.2.6 Time complexity

The time complexity of I-CSE mainly depends on the number n of variables, on the number e_a of a-ary operators (nodes) and on the maximum arity a of an a-ary sum or product expression in the system. $e_a + n$ is the size of the DAG created in Step 1, so that the time complexity of Step 1 is $O(e_a + n)$ on average if the node identifiers are maintained using hashing. In Step 2, the number i of performed intersections is quadratic in the number of sums (or products) in the DAG, i.e., $i = O(e_a^2)$. The number of calls to the UpdateCSlists procedure is, in the worst case, quadratic in the number of nodes, i.e., $O(e_a^2)$ (because the maximum size of CS_{list} is $e_a - 1$). The maximum number of $\subset$ operations performed by the UpdateCSlists procedure is $O(e_a)$. Every intersection, Complement or $\subset$ operation requires $O(a)$

on average using hashing (a worst-case complexity $O(a\,log(a))$ can be reached with sets encoded by trees/heaps). Thus, the time complexity of Step 2 is $O(a\,log(a)(e_a^2 + e_a^2 \times e_a)) = O(a\,log(a)\,e_a^3)$.

The worst-case complexity for Step 3 occurs when the number of CSs for each node is maximal $(e_a - 1)$ and the number of generated redundant expressions for each node is also maximal $(e_a - 2)$. In this case, the number of calls to the `Complement` and to the $\subset$ procedures in `Process` is $O(e_a^3)$. Step 4 is linear in the size of the final DAG and is $O(e_a + n + i)$. Overall, `I-CSE` is thus $O(n + a\,log(a)\,e_a^3)$.

Table 6.1 illustrates how the time complexity evolves in practice with the size of three representative scalable systems. The CPU times have been obtained with a processor Intel 2.40 `GHz`. The CPU time increases linearly in the size $e_a + n$ of the DAG for `Trigexp1` and `Katsura`. For `Brown`, we have $a = n$ and the time complexity seems to be less than $O(n\,log\,n\,e_a^3)$.

Table 6.1: Time complexity of `I-CSE` on three representative scalable systems of equations (see Section 6.4).

Benchmark	`Trigexp1`			`Katsura`			`Brown`			
Number n of variables	10	20	40	5	10	20	10	20	40	80
Number e_a of operators	46	96	196	15	55	208	10	20	40	80
`I-CSE` time in second	0.19	0.28	0.63	0.08	0.19	0.91	0.05	0.20	1.26	9.21

6.2.7 Maximal CSs shared by more than two expressions

As we have already mentioned, the `I-CSE` algorithm finds *all* the maximal CSs shared by any *pair* of expressions of the *original system*. However, `I-CSE` does not find a CS shared by three or more expressions that is not maximal in any intersection of two expressions.

Consider for instance the expressions $f_1 : x_1+x_2+x_3+x_4$, $f_2 : x_1+x_2+x_3+x_5$ and $f_3 : x_1+x_2+x_4+x_5$. The pairwise intersection (step 2) of the algorithm finds the maximal CSs shared by two expressions, i.e., $cs_{12} : x_1 + x_2 + x_3$ (shared by f_1 and f_2), $cs_{13} : x_1 + x_2 + x_4$ (shared by f_1 and f_3) and $cs_{23} : x_1 + x_2 + x_5$ (shared by f_2 and f_3). However, the maximal CS $cs_{123} : x_1 + x_2$ shared by the three expressions is not found nor replaced by the algorithm.

Maximal CSs common to three or more expressions that are not replaced by `I-CSE` appear *only* when *conflictive CSs have been found*. Recall that two CSs are in conflict when their intersection is not empty and one is not included in the other. If the intersection between two CSs (e.g., cs_{123}) corresponds to an expression of size larger than one (e.g., $cs_{12} \cap cs_{13} = x_1 + x_2$), then this intersection is the maximal CS shared by the expressions involving these two intersected CSs (e.g., f_1, f_2 and f_3). The number of involved expressions is greater than two.

The number of CSs in the worst case is exponential in the number of a-ary operators. An algorithm for finding them might simply consist in applying recursively the step 2 of `I-CSE` in the list of newly found CSs until a fixpoint is reached (i.e., until no new CS is found). In the example above:

1. the current step 2 finds cs_{12}, cs_{13} and cs_{23};
2. a second (new iteration) would add $cs_{123} = cs_{12} \cap cs_{13} = cs_{12} \cap cs_{23} = cs_{13} \cap cs_{23}$;
3. no further iteration is necessary to handle the new single CS cs_{123}.

We are afraid that CSs shared by k expressions ($k > 2$) are not useful in practice, and we will study and experiment this *advanced* version of `I-CSE` to confirm this result.

6.3 Implementation of I-CSE

I-CSE has been implemented using Mathematica version 6 [Wolfram, 2009]. Mathematica first automatically transforms the equations into a canonical form, where additions and multiplications are n-ary and where are performed reductions, i.e., factorizations by a constant. For instance, the expression $2x - y + x + z$ is transformed into $3x + (-y) + z$. The n-ary representation of equations is useful for the pairwise intersections of I-CSE (Step 2).

The solving algorithms are developed in the open source interval-based library in C++ called Ibex [Chabert, 2009; Chabert and Jaulin, 2009c]. A given benchmark is solved by a branch and prune process: the variables are bisected in a round-robin manner and contracted by constraint propagation (HC4 only, or 3BCID using HC4 (see Section 3.2.4.2)) and interval Newton. As mentioned at the end of Section 6.2.4, Ibex offers facilities to create two systems of equations in memory for which domains of variables are synchronized during the search of solutions.

I-CSE-B and I-CSE-NC

We have theoretically proven that the benefits of I-CSE lie in the additional pruning it permits and not only in a decrease of the number of operations. To confirm in practice this significant result, we have designed two variants of I-CSE that compute fewer CSs. I-CSE-B (Basic I-CSE) simply ignores the step 2 of I-CSE. The commutativity and associativity of + and × are not taken into account. Additive and multiplicative n-ary expressions are considered in a fixed binary form in which only a few subexpressions can be detected. For instance, the CS $x + y$ is detected in two expressions $x + y + z_1$ and $x + y + z_2$, but not in expressions $x + z_1 + y$ and $x + z_2 + y$.

I-CSE-NC (I-CSE with No Conflicts) completely exploits the commutativity and associativity of + and ×, but does not take into account conflictive CSs. I-CSE-NC lowers the worst case time complexity of I-CSE, but does not create all the CSs. If a given system does not contain CSs in conflict, I-CSE and I-CSE-NC return the same new system (with no redundant equations). In the example, I-CSE-NC does not create the redundant equation $v_3 = -1 + y + v_4$, so that the equation $v_4 = v_1 + y^3$ is not created either. The second (initial) equation finally becomes: $\frac{(x^2+y^3)\times v_5+14}{v_5} = 8$.

Existing CSE algorithms, including the one proposed in [Vu et al., 2004, 2009b], take place between I-CSE-B and I-CSE-NC in terms of number of detected useful CSs.

6.4 Experiments

Benchmarks have been taken in the first two sections (polynomial and non-polynomial systems) of the COPRIN page [Merlet, 2009]. The selected sample fulfills systematic criteria: every tested benchmark is an NCSP with a finite number of isolated solutions (no optimization); all the solutions can be found by the ALIAS system [Merlet, 2000] in a time comprised between one second and one hour; selected systems are written with the following primitive operators: +, -, ×, /, sin, cos, tan, exp, log, power. With these criteria, we have selected 40 benchmarks. The I-CSE algorithm detects no CS in 16 of them. There are also two more benchmarks (Fourbar and Dipole2) for which no test has finished before the timeout (one hour), providing no indication. 9 of the remaining 22 benchmarks are scalable, that is, can be defined with any number of variables. Table 6.2 provides information about the selected benchmarks. When there is no conflictive CS, I-CSE and I-CSE-NC return the same new system and there is no redundant

Table 6.2: Selected benchmarks. The columns yield the name of the benchmark, the number of solutions (#s), the number of variables (n), the number of useful CSs (#cs) found by `I-CSE-B`, `I-CSE-NC`, `I-CSE`, the number of redundant constraints created by `I-CSE` due to conflictive CSs (#rc).

Benchmark			I-CSE-B	ICSE-NC	I-CSE		Benchmark			I-CSE-B	ICSE-NC	I-CSE	
	#s	n	#cs	#cs	#cs	#rc		#s	n	#cs	#cs	#cs	#rc
6body	5	6	2	3	3	0	Katsura-20	7	21	90	90	90	0
Bellido	8	9	0	1	1	0	Kin1	16	6	13	13	19	3
Brown-7	3	7	3	7	21	24	Pramanik	2	8	0	15	15	0
Brown-7*	3	7	3	1	1	0	Prolog	0	21	0	7	7	0
Brown-30	2	30	26	53	435	783	Rose	16	3	5	5	5	0
BroyBand-20	1	20	22	37	97	73	Trigexp1-30	1	30	29	29	29	0
BroyBand-100	1	100	102	119	479	473	Trigexp1-50	1	50	49	49	49	0
Caprasse	18	4	6	7	11	2	Trigexp2-11	0	11	15	15	15	0
Design	1	9	3	3	3	0	Trigexp2-19	0	19	27	27	27	0
Dis-Integral-6	1	6	4	6	18	9	Trigonom-5	2	5	7	9	20	14
Dis-Integral-20	3	20	18	34	207	171	Trigonom-5*	2	5	7	6	6	0
Eco9	16	8	0	3	7	1	Trigonom-10	24	10	15	15	26	15
EqCombustion	4	5	7	8	11	1	Trigonom-10*	24	10	15	12	12	0
ExtendWood-4	3	4	2	2	2	0	Yamamura-8	7	8	5	10	36	48
Geneig	10	6	11	14	14	0	Yamamura-8*	7	8	5	1	1	0
Hayes	1	8	9	8	8	0	Yamamura-12	9	12	9	18	78	119
I5	30	10	3	4	10	5	Yamamura-12*	9	12	9	1	1	0
Katsura-19	5	20	81	81	81	0	Yamamura-16	9	16	13	26	136	224

constraints (#rc=0). The interval-based solver results will be the same.

For all the benchmarks, the CPU time required by `I-CSE` *(and variants) is often negligible and always less than 1 second.*

Remark. In the scalable benchmarks marked with a star (*), the equations have not been initially rewritten into the canonical form by `Mathematica` (see Section 6.3). This leads to fewer CSs, but these CSs correspond to larger subexpressions shared by more expressions, providing generally better results.

Tables 6.3 and 6.4 compare the CPU times required by `Ibex` to solve the initial system (Init) and the systems generated by `I-CSE-B`, `I-CSE-NC` and `I-CSE`. Table 6.3 reports the results obtained by a standard branch and prune approach with bisection, Newton and **HC4**. Table 6.4 reports the results obtained by a branch and prune approach with bisection, Newton and **3BCID** (using `HC4` as a refutation algorithm). Both tables report CPU times in seconds obtained on a 2.40 `GHz Intel Core 2` processor with 1 `Gb` of RAM, and the corresponding gain w.r.t. the solving of the original system. The time limit has been set to 3600 seconds. The tables also report the number of generated boxes (#Boxes) during the search. This corresponds to the number of nodes in the tree search and highlights the additional pruning due to `I-CSE`. The precision of solutions has been set to 10^{-8} for all the benchmarks. The parameter used by `HC4` has been set to 1% in Table 6.3. The parameters used by `HC4` and `3BCID` have been set to 10% in Table 6.4. We have put at the end of both tables the results corresponding to scalable benchmarks. To return a fair comparison between algorithms, we have selected for the scalable systems the instance with the largest number of variables n such that the solver on the original system finds the solutions in less than one hour. This number n is greater with `3BCID` (Table 6.4) than with only `HC4` (Table 6.3) because `3BCID` is generally more efficient than `HC4`.

Tables 6.3 and 6.4 clearly highlight that `I-CSE` is very interesting in practice. We observe a gain in performance greater than a factor 2 on 15 among the 24 lines (on both tables). The gain is of two orders of magnitude (or more) for 5 benchmarks with `HC4` (corresponding to 4 different systems) and for 10 benchmarks with `3BCID` (corresponding to 8 different systems).

`I-CSE` clearly outperforms the variants extracting fewer useful CSs, as shown on Table 6.3 (see `Brown-7`, `Dis-Integral-6`, `Broyden-Banded-20`) and Table 6.4 (see `Brown-7`, `Dis-Integral-6`, `Yamamura-12`,

Table 6.3: Results obtained with HC4 and interval Newton

Benchmark	Time in second				Time(Init) / Time			#Boxes		
	Init	ICSE-B	ICSE-NC	I-CSE	ICSE-B	ICSE-NC	I-CSE	Init	ICSE-NC	I-CSE
EqCombustion	>3600	26.1	0.35	**0.14**	>137	>10000	**>25000**	>1e+08	3967	**1095**
Rose	>3600	500	**101**	**101**	>7.2	**>35**	**>35**	>3e+07	**865099**	**865099**
Hayes	141	51.9	**15.7**	**15.7**	2.7	**9**	**9**	550489	**44563**	**44563**
6-body	0.22	**0.07**	**0.07**	**0.07**	**3.1**	**3.1**	**3.1**	4985	**495**	**495**
Design	176	65.2	**63.2**	**63.2**	2.7	**2.8**	**2.8**	425153	**122851**	**122851**
I5	>3600	>3600	**1534**	1565	?	**>2.3**	>2.3	>3e+07	**7e+06**	7e+06
Geneig	3323	2910	2722	2722	1.14	**1.22**	**1.22**	7e+08	4e+08	4e+08
Kin1	8.52	8.32	8.32	**8.01**	1.02	1.02	**1.06**	**905**	909	**905**
Pramanik	89.3	92.1	**84.9**	**84.9**	0.97	**1.05**	**1.05**	487255	**378879**	**378879**
Bellido	15.7	15.9	**15.6**	**15.6**	0.99	**1.01**	**1.01**	29759	**29319**	**29319**
Eco9	**23.9**	**23.9**	24	24.1	**1.00**	**1.00**	0.99	126047	117075	**110885**
Caprasse	**1.56**	1.81	1.68	2.16	0.86	0.93	0.72	8521	7793	**7491**
Brown-7*	500	350	**0.01**	**0.01**	1.42	**49500**	**49500**	6e+06	**95**	**95**
Dis-Integral-6	201	0.46	1.3	**0.03**	437	155	**6700**	653035	4157	**47**
ExtendWood-4	29.9	**0.03**	**0.03**	**0.03**	**997**	**997**	**997**	422705	**353**	**353**
Brown-7	500	350	30.7	**1.49**	1.42	16.1	**332**	6e+06	258601	**3681**
Trigexp2-11	1118	208	**56.2**	**56.2**	5.38	**19.9**	**19.9**	1e+06	**316049**	**316049**
Yamamura-8*	13	13.3	**0.75**	**0.75**	0.98	**17.3**	**17.3**	29615	**2161**	**2161**
Broy-Banded-20	778	759	261	**58.1**	1.03	2.98	**13.4**	172959	46761	**12623**
Trigonometric-5*	15.8	12.3	**1.49**	**1.49**	1.28	**10.6**	**10.6**	10531	**1503**	**1503**
Trigonometric-5	15.8	12.3	8.94	**6.97**	1.28	1.77	**2.27**	10531	7369	**5307**
Yamamura-8	13	13.3	44.6	**10.8**	0.98	0.3	**1.20**	29615	115211	**13211**
Katsura-19	**1430**	1583	1583	1583	0.90	0.90	0.90	**145839**	153193	153193
Trigexp1-30	**2465**	3244	3244	3244	0.76	0.76	0.76	1e+07	1e+07	1e+07

Trigonometric-10). In these cases, the gains in CPU time are significant. They are sometimes of several orders of magnitude. The few exceptions for which I-CSE is worse than its simpler variants give only a slight advantage to I-CSE-NC or I-CSE-B.

The number of boxes is generally decreasing from the left to the right of tables. This confirms our theoretical analysis that expects gains in filtering when a system has additional equations due to CSs. This experimentally proves that exploiting conflictive CSs is useful. This confirms an intuition shared by a lot of practitioners of partial consistency algorithms that redundant constraints are often useful because they allow a better pruning effect [Harvey and Stuckey, 2003]. Benchmarks Brown-30, Dis-Integral-20 and Yamamura-16, have been added at the end of Table 6.4 to highlight this trend: I-CSE produces a gain in performance of 3 orders of magnitude while it adds hundreds of redundant equations.

Most of the obtained results are good or very good, but four benchmarks observe a loss of performance lying between 20% and 42%: Caprasse with both strategies, and Pramanik, Geneig, Katsura with 3BCID. The loss in performance observed for Katsura-20 (10% or 42% according to the strategy) is due to the domains of the variables that are initialized to [0,1]. Without detailing, such domains imply that the pruning in the search tree is due to the evaluation (bottom-up) phase and not to the (top-down) narrowing phase of HC4-revise.

6.5 Perspectives

In other experiments (not detailed in this thesis) we observed the benefits of using I-CSE and symbolic-based extensions (described in Section 2.4.5) together. Consider for example a set of two expressions: $f = x_1y_1 + x_1y_2$ and $g = x_2y_2 + x_2y_3$. The I-CSE algorithm cannot detect any CS in the set. However if

Table 6.4: Results obtained with 3BCID using HC4 and interval Newton

Benchmark	Time in second				Time(Init) / Time			#Boxes		
	Init	ICSE-B	ICSE-NC	I-CSE	ICSE-B	ICSE-NC	I-CSE	**Init**	ICSE-NC	I-CSE
Rose	2882	5.17	**4.04**	**4.04**	557	**713**	**713**	4e+06	**5711**	**5711**
Prolog	38.5	60	**0.14**	**0.14**	0.64	**275**	**275**	4647	**11**	**11**
EqCombustion	0.42	0.37	**0.06**	**0.06**	1.35	7	7	427	**23**	**23**
Hayes	32.6	27.2	**5.67**	**5.67**	1.13	**5.7**	**5.7**	17455	**1675**	**1675**
Design	52	17.9	**13.3**	**13.3**	2.9	**3.9**	**3.9**	16359	**4401**	**4401**
I5	33.5	41.1	**17.9**	**17.8**	0.81	1.9	**1.9**	10619	4387	**4281**
6-body	0.14	**0.08**	0.1	0.1	**1.75**	1.4	1.4	173	**51**	**51**
Kin1	1.66	2.66	1.76	**1.23**	0.62	0.94	**1.35**	**85**	161	197
Bellido	10.3	10.4	**9.98**	**9.98**	1	**1.03**	**1.03**	4487	**4341**	**4341**
Eco9	**11.6**	**11.6**	12.4	13.2	**1**	0.94	0.88	6205	6045	**5749**
Pramanik	**73.8**	114	96.8	96.8	0.65	0.76	0.76	124663	**95305**	**95305**
Caprasse	1.96	2.51	2.5	2.92	0.74	0.78	0.67	1285	1311	**1219**
Geneig	**696**	1050	1050	1050	0.66	0.66	0.66	362225	**362045**	**362045**
Trigexp2-19	2308	2.23	**0.03**	**0.03**	1035	**77000**	**77000**	250178	**7**	**7**
Brown-7*	600	318	**0.01**	**0.01**	1.88	**60000**	**60000**	662415	**9**	**9**
ExtendWood-4	185	**0.03**	**0.03**	**0.03**	6167	6167	6167	669485	**35**	**35**
Dis-Integral-6	135	0.18	0.51	**0.03**	750	264	**4500**	86487	185	**7**
Brown-7	600	318	4.75	**0.22**	1.88	126	**2700**	662415	2035	**23**
Yamamura-12*	1751	1842	**1.01**	**1.01**	0.95	**1700**	**1700**	364105	**307**	**307**
Yamamura-12	1751	1842	31.1	**8.72**	0.95	56.3	**200**	364105	5647	**445**
Trigonometric-10*	1344	506	**19.4**	**19.4**	2.67	**69**	**69**	140512	**2033**	**2033**
Trigonometric-10	1344	506	156	**49.6**	2.67	8.62	**27**	140512	19883	**3339**
Broy-Banded-100	9.96	20.3	14.8	**8.21**	0.49	0.67	**1.21**	13	23	**11**
Trigexp1-50	**0.15**	0.19	0.17	0.17	0.79	0.88	0.88	**1**	**1**	**1**
Katsura20	**3457**	5919	5919	5919	0.58	0.58	0.58	**62451**	120929	120929
Brown-30	>3600	>3600	>3600	**22.9**	?	?	>150	>210021	>151527	**31**
Dis-Integral-20	>3600	>3600	>3600	**1.12**	?	?	**>3200**	>111512	>75640	**39**
Yamamura-16	>3600	>3600	681	**35.6**	?	>5	**>100**	>522300	96341	**919**

we transform the expressions using a Horner scheme we obtain the new set:

$$\begin{aligned} f_{h(x_1)} &= x_1(y_1 + y_2) \\ g_{h(x_2)} &= x_2(y_1 + y_2) \end{aligned}$$

Now, I-CSE can detect the CS $y_1 + y_2$.

We have implemented a simple Horner-based algorithm (motivated by the works of Ceberio, Granvilliers and Kreinovich [Ceberio and Granvilliers, 2001; Ceberio and Kreinovich, 2004]) that applies the Horner scheme in every *sum* expression f before performing I-CSE. The Horner scheme is applied to the factor appearing more times in f.

Example 34 *The result obtained by our algorithm when it is applied to different expressions is:*

$$\begin{aligned} f_1 = x_2x_3 + x_2x_1 &\rightarrow f_1' = x_2(x_3 + x_1) \\ f_2 = x_2(x_3 + x_1) + x_1(x_3 + x_1) &\rightarrow f_2' = (x_3 + x_1)(x_1 + x_2) \\ f_3 = x_1^3x_2 + x_1^2x_3 + x_1^2x_2x_3 &\rightarrow f_3' = x_1^3x_2 + x_1^2(x_3 + x_2x_3) \end{aligned}$$

The Horner scheme has been applied to the factors x_1, $(x_3 + x_1)$ and x_1^2 respectively. Observe that for f_3, x_1^2 is not factorized in the first term $x_1^3x_2$ because it would not lower the number of occurrences of x_1.

Results show that, for instance, in the Virasoro benchmark (also obtained from the COPRIN web page), using this method we obtain a gain in CPU time close to 4 w.r.t. I-CSE.

6.6 Conclusion

This chapter has presented the algorithm `I-CSE` for exploiting common subexpressions in numerical CSPs. A theoretical analysis has shown that gains in filtering can be expected only when CSs do not correspond to monotonic and continuous operators like x^3 or log. Contrarily to a belief in the community, this means that CSs can bring significant gains in filtering/contraction, and not only a decrease in the number of operations. These are good news for the significance of this line of research.

Experiments have been performed on 40 benchmarks among which 24 contain CSs. Significant gains of one or several orders of magnitude have been observed on 10 of them. `I-CSE` differs from existing CSEs in that it also detects conflictive CSs. As compared to `I-CSE-NC` (better than or similar to existing CSEs), the additional contraction involved by the corresponding redundant equations leads to improvements of one or several orders of magnitude on 4 benchmarks (`Brown`, `Dis-Integral`, `Yamamura` and, only for `HC4`, `BroyBanded`).

A future work is to compare our implementation based on the standard `HC4` algorithm (and the management of two systems), with the sophisticated propagation algorithm carried out on the elegant DAG-based structure proposed by Vu, Schichl and Sam-Haroud. However, our experimental results have underlined that the gain in contraction has a greater impact on efficiency than the time required to reach the fixpoint of propagation. Thus, we suspect that both implementations will show similar performances.

We have concluded that performing `I-CSE` as a preprocessing procedure can bring significant gains in filtering to a solving strategy based on the `HC4` algorithm. However this is not always true when the strategy is based on more sophisticated propagation algorithms (e.g., `Mohc`, `Box`). The problem is that replacing subexpressions by auxiliary variables can increase the *locality problem* due to the decomposition of constraints (see Section 3.6). One alternative is to extend the propagation algorithms for using the DAG representation of the system. Another alternative would be to modify the `Revise` procedures of the propagation algorithms to take into account both the initial and the generated system (with CSs).

In the next chapter we present a new constraint propagation algorithm focusing on subsystems of size k.

Chapter 7

A Filtering Algorithm Using Well-constrained Subsystems

Contents

In this chapter we describe a new constraint propagation algorithm called `Box-k`. `Box-k` contracts domains of variables involved in well-constrained subsystems of size k (k constraints and k variables). Well-constrained subsystems act as *global constraints* that can bring additional filtering w.r.t. interval Newton (described in Section 3.2.1) and constraint propagation algorithms like `HC4` (described in Section 3.2.3.4) or `Box` (described in Section 3.2.3.5).

(A part of the material presented in this chapter is published in [Araya et al., 2009c,d].)

7.1 Introduction: From decomposable to sparse systems

Neveu, Trombettoni, Bliek, Jermann and Chabert have proposed in [Bliek et al., 1998; Neveu et al., 2006, 2005] an algorithm to treat decomposable systems. Inter-Block Backtracking (`IBB`) solves nonlinear systems of constraints that have been first decomposed into a sequence of *irreducible* blocks/subsystems [Ait-Aoudia et al., 1993]. When the blocks are sufficiently small, `IBB` outperforms classical solvers in several orders of magnitude. This is due to the fact that the classical solvers have no knowledge about the structure of the system implying possible bad selection of variables to bisect (the bisections performed by `IBB` depend on the sequence of subsystems).

`IBB` treats each block/subsystem in the order provided by the sequence. The algorithm interleaves contraction steps (performed by `HC4` and interval Newton) and bisections inside the block until atomic boxes (solutions) are obtained. Choice points are then made: the variables of the block are replaced by one of the atomic boxes, i.e., they are considered constants/parameters in subsequent blocks.

Let us write, in Algorithm 19, a simple variant of `IBB`.

Algorithm 19 `IBB`(**in**: $P = (C, X, [B_0]), \epsilon$, **out**: L)

1: L $\leftarrow \{[B_0]\}$
2: **while** There exist non-atomic boxes in L **do**
3: Select and delete a non-atomic box $[B]$ from L
4: $P^* \leftarrow$ `Parameterize_Atomic_Intervals`$(P, [B])$
5: $P^{k\times k} \leftarrow$ `Get_Subsystem`(P^*)
6: $S \leftarrow$ `Solve`$(P^{k\times k}, \epsilon)$ /* using a branch and prune based strategy */
7: $L \leftarrow L \cup S$
8: **end while**

L maintains a set of boxes that will be narrowed and split until becoming atomic. L can be seen as the set of leaves of the search tree. At the beginning L is initialized with the initial domains $[B_0]$. While there exist non atomic boxes in L, such a box $[B]$ in L is processed by the `Parameterize_Atomic_Intervals` procedure. The procedure replaces each variable x_i associated to an atomic interval in the system P by the atomic constant $[x_i]$, creating a new system P^*. (Constraints that are not related with any variable after the replacement, are excluded from P^*).) Then, the `Get_SubProblem` procedure uses the new system for finding a *well-constrained subsystem* (see Definition 23) of k constraints related to exactly k variables. The `Solve` procedure uses a branch & prune strategy (see Section 3.1) for finding the set of solutions/boxes S of the subsystem $P^{k\times k}$ with a precision ϵ. Remark that only k of the n intervals of the boxes in S are reduced to atomic intervals by the `Solve` procedure. Thus, non-atomic boxes in S are still processed by `IBB` until they becomes atomic or are eliminated.

The main difference with the original `IBB` is that Algorithm 19 performs the decomposition *on the fly* (line 5). Whereas, `IBB` simply picks the next block in the precomputed sequence of subsystems.

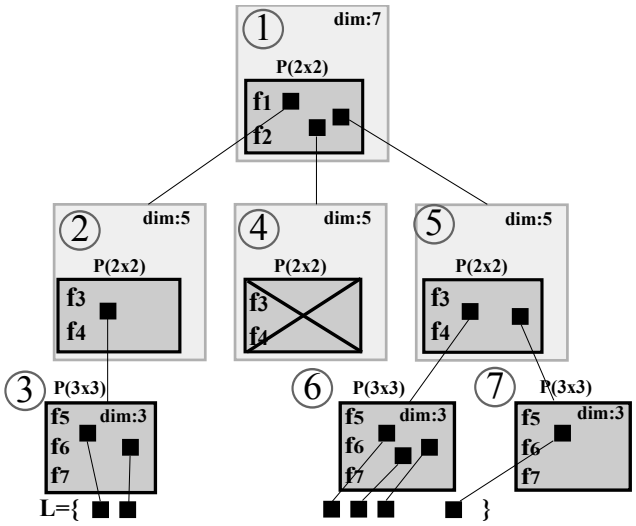

Figure 7.1: Example of a tree search performed by the `IBB` algorithm.

Consider the system $P = (X, C, [B])$, where the set of constraints C is given by:

$$f_1(x_1, x_2) = 0 \qquad f_5(x_1, x_3, x_4, x_5, x_6, x_7) = 0$$
$$f_2(x_1, x_2) = 0 \qquad f_6(x_3, x_4, x_5, x_6, x_7) = 0$$
$$f_3(x_1, x_3, x_4) = 0 \qquad f_7(x_4, x_5, x_6, x_7) = 0$$
$$f_4(x_1, x_2, x_3, x_4) = 0$$

and $X = \{x_1, ..., x_7\}$. In Figure 7.1 we can observe a trace of a tree search performed by the algorithm on P. All the rectangles represent a subproblem of P. The top left number indicates the order in which the subproblems are treated. The top right number indicates the dimension of the box associated to the problem (without taking into account the atomic intervals). Consider, for instance, the root problem (rectangle 1). It corresponds to the initial problem P. When the algorithm executes the `GetSubsystem` procedure in $P = (X, C, [B])$, it obtains a 2×2 well-constrained subsystem $P^{2\times2} = (X', C', [B])$ (represented by the rectangle inside the rectangle 1). The system $P^{2\times2}$ is given by:

$$\begin{aligned} f_1(x_1, x_2) &= 0 \\ f_2(x_1, x_2) &= 0 \end{aligned}$$

The solving of this subproblem returns three solutions (the black squares in the figure). Each solution/box $[B']$ generates a new subproblem $P' = (X, C, [B'])$ (corresponding to rectangles 2,4 and 5 in the figure). Consider for example the first solution/box $(\dot{x_1}, \dot{x_2}, [x_3], ..., [x_8])$, with $\dot{x_1}$ and $\dot{x_2}$ the atomic intervals obtained by the solver. Continuing with this solution, the `Parameterize_Atomic_Intervals` procedure will create the following new system P^*:

$$\begin{aligned} f_3^*(x_3, x_4) = f_3(\dot{x_1}, x_3, x_4) = 0 \qquad & f_6(x_3, x_4, x_5, x_6, x_7) = 0 \\ f_4^*(x_3, x_4) = f_4(\dot{x_1}, \dot{x_2}, x_3, x_4) = 0 \qquad & f_7(x_4, x_5, x_6, x_7) = 0 \\ f_5(x_1, x_3, x_4, x_5, x_6, x_7, x_8) = 0 & \end{aligned}$$

The `GetSubsystem` procedure then finds the following well-constrained subsystem $P^{2\times2}$:

$$\begin{aligned} f_3^*(x_3, x_4) &= 0 \\ f_4^*(x_3, x_4) &= 0 \end{aligned}$$

Then, the subsystem is solved and only one solution is found $(\dot{x_3}, \dot{x_4})$ leading to the box $(\dot{x_1}, ..., \dot{x_4}, [x_5], ..., [x_8])$. Observe that each subproblem is treated as a new independent one, i.e., in each of them the algorithm searches for a new square subsystem and solves it.

When all the boxes in L become atomic and satisfy all the equations of the system P, L becomes the set of solutions of the problem.

In this chapter, we propose a generalization of Algorithm 19 called `Box-k`. Our algorithm also manages $k \times k$ well-constrained subsystems. However, it treats subsystems containing thick constants/parameters. Our algorithm solves a given subsystem of size k with a rough precision before returning the hull of the thick solutions, thus obtaining a contraction on the involved variables. As a result, contrarily to `IBB`, `Box-k` is a propagation algorithm embedded in a classical branch & prune method.

The contraction algorithm is presented in Section 7.3. It enforces a new kind of consistency presented in Section 7.2. In Section 7.4, we present *multidimensional splitting* (in short `multisplit`). A `multisplit` consists in splitting several variable domains simultaneously in order to perform a choice point in the search (i.e., the current box is split in several subboxes). `multisplit` generalizes line 7 of Algorithm 19, where `IBB` adds in L the solutions/boxes of the solved subsystem.

Promising experiments, presented in Section 7.5, highlight the benefits of our approach for decomposed and structured systems.

7.2 Box-k partial consistency

As explained in Section 3.2.3.5, the Box-consistency yields an outer approximation/box of 1×1 subsystems (c, x). The Box-k-consistency introduced in this thesis generalizes Box-consistency by yielding an outer approximation of $k \times k$ subsystems.

Definition 21 *(**Subsystem**) Consider the NCSP $P = (X, C, [B])$. $C' \subset C$ (with $C' : F'(X') = 0$) is a subset of constraints involving the variables in $X' = (X_I \cup X_O) \subset X$ (X_I and X_O are two disjoint sets of variables).*

$P' = (X_O, C'', [X_O])$, with a set of equations $C''(X_O) : F''(X_O) = 0$, is a **subsystem** *of P if*

$$F''(X_O) = F'([X_I], X_O)$$

Observe that the variables in X_I have been replaced by their domains.

- *X_I is the set of* **input parameters** *of the subsystem P' and*
- *X_O is the set of* **output variables** *of the subsystem P'.*

Definition 22 (Box-k-consistency) *Consider the subsystem of equations $C : F(X_O) = 0$ with $|X_O| = k$. C is box-k-consistent in the box $[B] = [X_I] \times [X_O]$ ($[X_I]$ is the box related to the input parameters of C) if there exists a k-box $[X_O^\epsilon]$ of size 1 u.l.p. on every face of the k-box $[X_O]$ such that each constraint $[f](X_O) = 0$ in C is satisfied, i.e.:*

$$0 \in [f]([X_O^\epsilon])$$

If a box-k-consistent subsystem has an empty set of input parameters, note that this subsystem is also global hull consistent [Cruz and Barahona, 2001]. Thus, like for the standard box-consistency (i.e., box-1-consistency), the presence of input parameters makes box-k-consistency weaker than global hull-consistency.

Figure 7.2-left shows an example of a 2×2 subsystem. The outer box is box-consistent since it optimally approximates the solution set of constraints c_1 and c_2 individually. The inner box is box-2-consistent since it optimally approximates the set of six "thick" solutions to both constraints. Constraints are thick because the input parameters (e.g., w_1, w_2, w_3) are replaced by intervals.

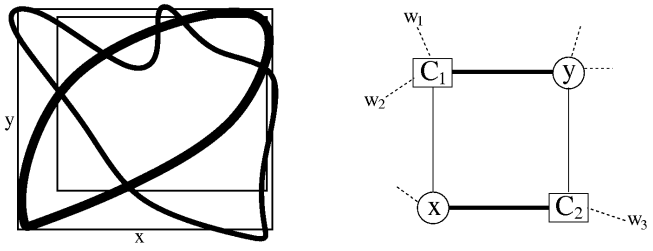

Figure 7.2: Illustration of Box-2-consistency

Partial consistencies are generally defined modulo a precision ϵ that is used in practice by the corresponding algorithm to reach a fixpoint earlier. ϵ must then replace 1 u.l.p. in the previous definitions.

7.2.1 Benefits of Box-k-consistency

The following example theoretically shows that a contraction obtained by a $k \times k$ subsystem may be stronger than contraction on 1×1 subsystems and on the whole $n \times n$ system performed by an interval Newton. Consider the system $\mathcal{S} = (\{x, y, z\},\ \{x - y = 0,\ x + y + z = 0,\ (z-1)(z-4)(2x+y+2) = 0\},\ \{[-10^6, 10^6], [-10^6, 10^6][-10, 10]\})$.

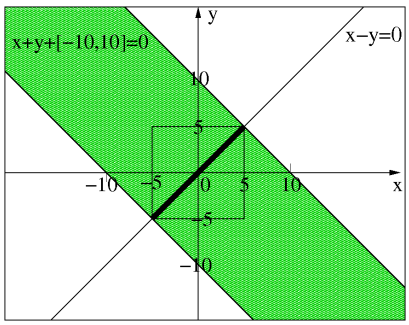

Figure 7.3: Illustration of a subsystem of size 2, with $[z] = [-10, 10]$ as input parameter. {[-5,5],[-5,5]} is box-2-consistent w.r.t. the 2 constraints $x - y = 0$ and $x + y + [z] = 0$.

Running `Box` and interval Newton on $\mathcal{S}$ does not filter the box. Achieving Box-2-consistency on the 2×2 subsystem ({x,y}, {x-y=0, x+y+z=0}) narrows the intervals of x and y to $[-5, 5]$ as shown in Figure 7.3. Also, if branching was used to find solutions, only two bisections (choice points) would be necessary to isolate the 3 solutions $\{(\frac{-2}{3}, \frac{-2}{3}, \frac{4}{3}), (-0.5, -0.5, 1), (-2, -2, 4)\}$. We should highlight that Newton on the whole system does not contract the box because it contains several solutions, whereas Newton on the 2×2 subsystem does because it contains only one (thick) solution (segment in bold). Of course, this small example is didactic. Experiments described in Section 7.5 show larger and nonlinear instances highlighting the benefits of structural partial consistencies over stronger partial consistencies like 3B-consistency [Lhomme, 1993].

7.2.2 Achieving Box-k-consistency in well-constrained subsystems of equations

Enforcing Box-k-consistency in every subsystem of given size k is too time-consuming and counter-productive in practice. The number of subsets of k variables in a system with n variables is high and one needs to consider only promising subsystems.

We have thus used several criteria to reduce the number of subsystems that are candidate. We first select subsystems with only equations (no inequalities) because equations bring a great reduction of the search space and have nice properties. To understand these properties, we have to pay attention to systems that admit a finite number of solutions. These systems contain n variables but also the same number n of independent equations (additional inequalities can reduce the number of solutions). Also, the corresponding *bipartite constraint graph* verifies the following structural/graph property [Ait-Aoudia et al., 1993].

Definition 23 *Let $\mathcal{S}$ be a system of n independent equations constraining n variables. The vertices of the* **bipartite constraint graph** *G corresponding to $\mathcal{S}$ are the n variables and the n equations, and edges connect one equation to its involved variables.*

The system of equations $\mathcal{S}$ is **(structurally) well-constrained** *if its constraint graph G has a perfect matching [Dulmage and Mendelsohn, 1958].*

For instance, Figure 7.2-right, page 138 shows the perfect matching (bold-faced edges) of the corresponding subgraph. This structural well-constriction can be viewed as a necessary condition to obtain a finite set of solutions. It appears that interval Newton also requires this condition (while it is of course not sufficient) for contracting a box. Indeed, if the system is not structurally well-constrained, the Jacobian matrix will necessarily be singular [Ait-Aoudia et al., 1993]. Our subsystems fulfill this condition because Interval Newton is used by our new Box-k-Revise procedure (see Section 7.3) to achieve faster a box-k-consistent subsystem. (Also, the time complexity of interval Newton is cubic in the number of variables, so that it is sometimes intractable to apply it to very large systems. Instead, we could use interval Newton only inside subsystems.)

We finally require our subsystems to be connected for performance considerations. Indeed, if a given subsystem of size k contained several disconnected components of size at most k' ($k' < k$), we could make it box-k-consistent by achieving box-k'-consistency in every component.

To sum up, restricting the subsystems to well-constrained and connected subgraphs of equations has two virtues. First, it allows a strong filtering in specific subparts of the system, which is useful for sparse systems or for (globally) under-constrained ones, e.g., systems mixing equalities and inequalities. Second, it allows the use of an interval Newton to faster contract the subsystem.

7.3 Contraction algorithm using well-constrained subsystems as global constraints

Instead of contracting all the well-constrained subsystems of given size k, we have designed an AC3-like propagation that manages selected subsystems of different sizes: subsystems of size 1 but also well-constrained subsystems of larger size. Well-constrained subsystems are thus similar to *global constraints* [Lebbah et al., 2005; Régin, 1994] that can be defined by the user or automatically (see Section 7.5).

All the subsystems are first put into a propagation queue and revised in sequence. When a variable domain is reduced more than a ratio ρ_{propag}, all the subsystems involving this variable are pushed into the queue, if they are not already in it. This propagation process is just specialized by the revise procedure used for contracting the subsystems of size greater than 1 and detailed below.

7.3.1 The `Box-k` revise procedure

The revise procedure, described in Algorithm 20 is based on a branch & prune method limiting the bisection to the k (output) variables X of the subsystem, and using a *breadth-first* search. At the end of this local tree search, the current box is replaced by the hull of the leaves of the local tree. The algorithm `Box-k-Revise` is a generic procedure that achieves a box-k-consistent subsystem. The procedure manages a list L of nodes that are leaves of the local tree. A leaf l in L has three significant components: $l.box$ designs the (n-dimensional) search space associated to the node; $l.precise$ is a boolean stating whether $l.box$ has reached the precision ϵ in all the dimensions (ϵ also yields the precision of the global solution); $l.certified$ is a boolean asserting whether $l.box$ contains a unique solution. The `box` parameter is the current global box (search space) when the revise procedure is called.

A combinatorial process (tree search) is performed by the `while` loop. At every iteration, one leaf in L, which is not *precise* and not *certified*, is selected, bisected and the two new sub-boxes are contracted. The search ends if all the leaves are tagged as *certified* or *precise* or if a limit τ_{leaves} in the number of leaves

Algorithm 20 `Box-k-Revise` (**in-out** L, `box`; **in** X, C, ϵ, `subContractor`, τ_{leaves}, $\tau_{\rho_{io}}$)

```
UpdateLocalTree(L, box, X, C, ε, subContractor)
L' ← {l ∈ L s.t. ¬l.certified and ¬l.precise and ProcessLeaf?(l,X,C,τ_ρio)}
while 0 < L'.size and L.size < τ_leaves do
  l ← L'.front() /* Select a leaf in breadth-first order */
  (l1, l2) ←bisect(l, X)
  contract(l1, subContractor, X, C, ε)
  contract(l2, subContractor, X, C, ε)
  if l1.box ≠ ∅ then L.pushBack(l1) end if
  if l2.box ≠ ∅ then L.pushBack(l2) end if
  L.remove(l)
  L' ← {l ∈ L s.t. ¬l.certified and ¬l.precise and ProcessLeaf?(l,X,C,τ_ρio)}
end while
box ←hull(L) /* Outer approximation of the union of all the boxes l.box, l ∈ L */
```

is reached. τ_{leaves} limits the memory storage requirement (see Section 7.3.5) and allows one to quickly propagate the obtained reductions to the other subsystems.

A leaf is simply selected in breadth-first order. We first tried a more sophisticated heuristic function for selecting a "large" box on the border of the hull of the different leaves. The idea was to maximize the gain in volume on the current global box in case the selected leaf would be eliminated by filtering. This multi-dimensional generalization of the `BoxNarrow` algorithm (see Section 3.2.3.5, page 50) has been discarded because it did not bring a significant gain in performance.

Algorithm 21 `contract`(**in-out** l; **in** `subContractor`, X, C, ϵ)

```
if ¬l.precise then
  if ¬l.certified then subContractor(l.box) end if
  if l.box ≠ ∅ and I-Newton(l.box,X) then l.certified ← true end if
  if maxDiameter(l.box) < ε then l.precise ← true end if
end if
```

The procedure `contract` is mainly parameterized by the contraction procedure `subContractor` (`HC4` [Benhamou et al., 1999] or `3BCID` [Trombettoni and Chabert, 2007] in our experiments). The scope C of `subContractor` is the considered k-set of equations. After a call to `subContractor`, an interval Newton limited to the $k \times k$ subsystem is launched. If Newton certifies a unique solution in a leaf, `I-Newton` contracts *l.box* and returns *true* so that this leaf is tagged as *certified*.

7.3.2 The `S-kB-Revise` variant

`S-kB-Revise` is the name of a variant of `Box-k-Revise` for which the entire system is used in the `contract` procedure. That is, the scope C of `subContractor` includes the whole n-set of constraints, instead of the k-set of constraints attached to the subsystem. (However, the propagation queue of `subContractor` is incrementally initialized with the k constraints.) With `S-kB-Revise`, the k-set of constraints in the subsystem is mainly used by interval Newton. This variant brings additional filtering, but at a higher cost.

7.3.3 Reuse of the local tree (procedure `UpdateLocalTree`)

A simpler version of Algorithm 20 did not call the `UpdateLocalTree` procedure and simply initialized the list L with the current box. However, instead of performing an intensive search effort in only one subsystem, we preferred to quickly propagate the obtained reductions to the other subsystems. Therefore the `UpdateLocalTree` procedure reuses the local tree (i.e., its leaves) that has been saved in a previous call to Algorithm 20. Every leaf in the current list L is just updated by intersection with the current box and filtered with `subContractor`.

Algorithm 22 `UpdateLocalTree` (**in-out** L; **in** `box`, X, C, ϵ, `subContractor`)

```
if L = ∅ then
  L ← {Leaf(box)} /* Initialize the root of the local tree with the current box */
else
  for all l ∈ L do
    /* Update and contract every leaf of the stored local tree */
    if l.box ≠ (l.box ∩ box) then
      l.box ← l.box ∩ box
      contract(l, subContractor, C, ϵ))
      if l.box = ∅ then L.remove(l) end if
    end if
  end for
end if
```

In fact, the leaves of the local trees are also maintained in the global search tree. To do so, the list L is implemented as a *backtrackable* data-structure updated in case of backtracking. It avoids redoing the same job in the subsystems several times, in particular when the *multisplit* splitting heuristic is chosen (see Section 7.4).

7.3.4 Lazy handling of a leaf (procedure `ProcessLeaf?`)

Our first experiments have shown us that handling a leaf in a local tree, i.e., bisecting it and contracting the two sub-boxes, was often counterproductive. We have then defined an input/output ratio ρ_{io} that decides whether a given leaf related to the subsystem $P^{k\times k}(X_O, C, [B])$ must be handled in the local tree.

The function `ProcessLeaf?` calculates ρ_{io} in the subsystem $P^{k\times k}$. If the ratio between the impact of the input parameters and the impact of the output variables is less than a threshold, then τ_{io} the subsystem will be revised. The computation of ρ_{io} is given by:

$$\rho_{io}(P^{k\times k}, X_I) = \frac{max_{x_i\in X_I}(\texttt{smear}(x_i, C, [B]))}{max_{x_i\in X_O}(\texttt{smear}(x_i, C, [B]))}$$

where X_I is the set of input parameters of $P^{k\times k}$. The impact of each variable x_i on the subsystem is calculated using the well-known smear function (described in Section 3.3). Consider a set of constraints defined by $C : F(X) = 0$. The smear function, associated to a variable x_i in X, is given by is given by:

$$\texttt{smear}(x_i, C, [B]) = \max_{f\in F}\left(\left|\left[\frac{\partial f}{\partial x_i}\right]([B])\right| \times \texttt{Diam}([x_i])\right)$$

The denominator of ρ_{io} can be directly explained by it: output variables (X_O) with a great smear evaluation (implying a small ratio ρ_{io}) often lead to a great contraction when they are bisected inside

the local subsystem tree. Desiring a small impact of the input parameters (X_I) is less intuitive. We understand that large input domains generally lead to large output domains (i.e., leaf boxes) in the subsystem and thus yields a poor reduction. The same argument holds in fact for the derivatives of functions. To illustrate this point, let us take a subsystem of size 1 like $0.001\,y + x^2 - 1 = 0$ (x is the output variable; $[x] = [y] = [-1, 1]$) having $\rho_{io} = \frac{0.002}{4} = 0.0005$. After one bisection on x, the subsystem contraction leads to a very small interval for x. A large interval would be obtained for x if the considered subsystem was $y + x^2 - 1 = 0$ with $\rho_{io} = \frac{2}{4} = 0.5$.

7.3.5 Properties of the revise procedure

The following proposition formalizes the correctness, the memory and time complexities of the procedure `Box-k-Revise`.

Proposition 26 *Let $P' = (X', C', [B])$ (with $|X'| = |C'| = k$) be a subsystem of $P = (X, C, [B])$. The* `Box-k-Revise` *procedure, called with $\tau_{leaves} = +\infty$ and $\tau_{\rho_{io}} = +\infty$, is able to enforce the box-k-consistency in P'.*

Let Diam be the largest interval diameter in $[B]$. Let d be $log_2(\frac{Diam}{\epsilon})$, the maximum number of times a given interval must be bisected to reach the precision ϵ. [1]

The memory complexity of `Box-k-Revise` *is $O(k\,\tau_{leaves})$.*

The number of calls to `subContractor` *is $O(k\,d\,\tau_{leaves})$.*

Proof 14 *The correction is based on the combinatorial process performed by the procedure* `Box-k-Revise`*. Called with $\tau_{leaves} = +\infty$ and with the subsystem P', the procedure computes all the atomic k-dimensional boxes, related to X', of precision ϵ before returning the hull of them, thus achieving roughly (i.e., assuming that the input parameters are atomic) the global consistency of C'.*

The memory complexity comes from the breadth-first search that must store the $O(\tau_{leaves})$ leaves of the local tree. The revise procedure works with n-dimensional boxes but, in order to save memory, stores at the end only k intervals of a $k \times k$ subsystem.

The number of calls to `subContractor` *is bounded by the number of nodes in the local search tree. The number of leaves of this tree is τ_{leaves} (corresponding to living boxes that can contain solutions) plus the number of dead leaves eliminated by filtering. For any living leaf l, the number of nodes created in the tree to reach l is at most $2 \times d \times k$ since the root must be at most bisected d times in all its k dimensions. Although numerous such internal nodes are "shared" by several living leaves, this bounds the number of calls to a sub-filtering operator with $O(k\,d\,\tau_{leaves})$.* □

Another property allows us to better understand the gain in contraction obtained by the `S-kB-Revise` variant (see Section 7.3.2).

Proposition 27 *Consider a propagation algorithm calling* `S-kB-Revise` *on all the subsystems of size k in a given NCSP P.*
This algorithm computes the $(k+2)B$-consistency of P.

The kB-consistency, introduced by Lhomme [Lhomme, 1993] (see Section 3.2.4.1), is a strong partial consistency related to the k-consistency (in finite-domain CSPs) restricted to the bounds of intervals.

[1] d generally falls between 20 and 60 in systems occurring in practice.

7.4 Multidimensional splitting

It turns out that the `Box-k-Revise` procedure has not only a contraction effect, but also provides a new way to make choice points, that is, to build the (global) search tree. This new splitting strategy is called *multidimensional splitting* (in short *multisplit*).

Definition 24 *Consider a $k \times k$ subsystem P' defined inside an NCSP $P = (X, C, [B])$. Consider a set S of m boxes associated to P' such that S contains all the solutions to P, and the m boxes obtained by projection on P' of the boxes in S are pairwise disjoint.*

A **multisplit** *of dimension k consists in splitting the search space $[B]$ into the m boxes in S.*

In practice, the m boxes correspond to the leaves of a subsystem local tree. At the end of a `Box-k` propagation, our solving strategy makes a choice between a classical bisection and a multisplit. If all the subsystems have a ratio ρ_m larger than a user-defined threshold τ_m, then a standard bisection is performed. Otherwise, we multisplit the subsystem with the smallest ratio ρ_m, i.e., we replace the current box by the set L of m leaves associated to its local tree.

$$\rho_m = \frac{\sum_{l \in L} \texttt{Volume}(l)}{\texttt{Volume}(Hull(L))}$$

Multisplit generalizes a procedure used by `IBB`. `IBB` performs a multisplit once it finds the m solutions (i.e., atomic boxes) in a given block. The difference here is that a multisplit may occur with *non* atomic boxes whose size has not reached the required precision.

7.5 Experiments

The `Box-k` based propagation algorithm has been implemented in the `Ibex` open source interval-based solver in `C++` [Chabert, 2009; Chabert and Jaulin, 2009a]. The variant with multisplit (`msplit`) performs a multisplit of a subsystem with the minimum ratio ρ_m, provided that $\rho_m < \tau_m$=0.99. All the competitors are also available in the same library, making the comparison fair.

7.5.1 Experiments on decomposed benchmarks

Ten *decomposed* benchmarks, described in [Neveu et al., 2006, 2005], appear in Table 7.1. They have been previously decomposed by equational algorithms (*eq*) like maximum-matching, or by more sophisticated geometrical algorithms (*geo*). They are challenging for general-purpose interval methods, but can efficiently be solved by `IBB` [Neveu et al., 2006, 2005] (described in sections 3.4.3 and 7.1).

Experimental protocol

Every `Box-k` based strategy has been tuned with 6 different sets of parameter values: $\tau_{\rho_{io}}$ is 0.01, 0.2 or 0.8 (0.01 is always the best value on decomposed systems); the precision ρ_{propag} used in the `HC4` propagation is 1% or 10%; All the other parameters have been empirically fixed: the precision ρ_{propag} in the `Box-k` propagation is always 10%; the maximum number τ_{leaves} of leaves inside a subsystem tree is

Benchmark	n	#sols	HC4	Box	3BCID	IBB	Box-k(HC4)		Box-k(3BCID)	
								msplit		msplit
Chair(eq)	178	8	>3600	>3600	>3600	**0.27**	>3600	16.5	>3600	**0.52**
1x15,1x13,1x9,5x8,3x6,...								575		15
Latham(eq)	102	96	>3600	>3600	39.9	**0.17**	**0.94**	1.35	1.5	1.08
1x13,1x10,1x4,25x2,25x1					587		839	199	991	189
Ponts(eq)	30	128	33.4	33.4	1.89	**0.59**	6.85	8.19	0.79	**0.71**
1x14,6x2,4x1			20399	20399	357		783	231	307	231
Ponts(geo)	38	128	44.1	44.1	2.6	**0.16**	2.01	0.31	1.45	**0.39**
13x2,12x1			18363	18363	685		6711	767	6711	767
Sierp3(geo)	124	198	>3600	>3600	77.5	**0.62**	49.0	**1.38**	52.5	1.77
44x2,36x1					1727		84169	1513	84169	1513
Star(eq)	46	128	>3600	>3600	4.9	**0.05**	35.6	**0.12**	44.0	0.26
3x6,3x4,8x2					283		44195	263	44023	263
Tangent(eq)	28	128	77	77	2.1	**0.08**	1.74	**0.08**	1.87	0.14
1x4,10x2,4x1			390903	390903	753		12027	255	12235	255
Tangent(geo)	42	128	–	–	7.38	**0.08**	0.80	0.19	0.80	**0.19**
2x4,11x2,12x1					859		1415	251	1407	251
Tetra(eq)	30	256	1281	1281	12.3	**0.63**	33.6	1.06	13.57	**0.76**
1x9,4x3,1x2,7x1			607389	607389	1713		4619	483	2243	483
Sierp3(eq)	see Section 7.5.2					>5000	see Section 7.5.2			

Table 7.1: Experimental results on `IBB` benchmarks.

10; the number of slices of `3BCID` in `Box-k(3BCID)` is 10. To be fair, the parameters of the competitor algorithms have been tuned so that 8 trials have been performed for `Box` and `HC4`, and 16 trials have been run for `3BCID`. For all the tests, the Newton ceil (size of maximum diameter under which interval Newton is run) is 10, and the same variable order is used in a round-robin strategy (except for `IBB` and for `Box-k` with multisplit).

The subsystems given to our `Box-k` propagation are defined automatically. The irreducible blocks produced by the `IBB` decomposition simply become the well-constrained subsystems handled by `Box-k-Revise`.

Results

Table 7.1 shows results on `IBB` benchmarks. The first 3 columns include the name of the system, its number n of variables and its number of solutions. The next three columns yield the CPU time (above) and the number of boxes, i.e., choice points (below), obtained on an `Intel 6600 2.4 GHz` by existing strategies based on `HC4`, `Box` or `3BCID` followed by interval Newton (between two bisections selected in a round-robin way for the variable selection). The last four columns report the results obtained by our algorithms on the same computer: `Box-k-Revise` parameterized by `subContractor=HC4` or `subContractor=3BCID`, with multisplit (`msplit`) or without. To be the closest to `IBB`, `Box-k-Revise`, and not the `S-kB-Revise` variant, is used by our constraint propagation algorithm.

Strategies based on `HC4`, `Box` and `3BCID` followed by interval Newton are not competitive at all with `Box-k` and `IBB` on the tested decomposed systems. The comparison of `Box-k` against `IBB` is very positive because

the CPU times reported for IBB are really the best that have never been obtained with any variant of this dedicated algorithm. Also, no timeout is reached by Box-k+multisplit and IBB is on average only twice faster than Box-k(3BCID) (at most 6 on Latham). As expected, the results confirm that multisplit is always relevant for decomposed benchmarks. For the benchmark Sierp3(eq) (the fractal Sierpinski at level 3 handled by an equational decomposition), an equational decomposition makes appear a large irreducible 50×50 block of distance constraints. This renders IBB inefficient on it (timeout).

7.5.2 Experiments on structured systems

Eight *structured* systems appear in Table 7.2. They are scalable chains of constraints of reasonable arity [Merlet, 2009]. They are denoted *structured* because they are not sufficiently sparse to be decomposed by an equational decomposition, i.e., the system contains only one irreducible block, thus making IBB pointless. A brief and manual analysis of the constraint graph of every benchmark has led us to define a few well-constrained subsystems of reasonable size (between 2 and 10). In the same way, we have replaced the 50×50 block in Sierp3(eq) by 6×6 and 2×2 Box-k subsystems.

Benchmark	n	#sols	**HC4**	**Box**	**3BCID**	**S-kB(HC4)**		**S-kB(3BCID)**		**Box-k-Revise**	
							msplit		**msplit**	HC4	3BCID
Bratu	60	2	58	626	48.7	47.0	**33.0**	135	126	86.4	96.2
29x3			15653	13707	79	39	17	43	25	125	129
Brent	10	1015	1383	127	**17.0**	28.5	20.2	44.9	31.0	20.8	34.9
2x5			7285095	42191	9849	2975	4444	4585	1309	5215	4969
BroydenBand	20	1	>3600	0.17	**0.11**	0.45	0.15	0.91	0.31	0.30	0.28
1x6,3x5				1	21	4	19	17	3	7	3
BroydenTri	30	2	1765	**0.16**	0.25	0.22	0.24	0.39	0.29	0.19	0.23
6x5			42860473	63	25	11	19	9	3	19	17
Reactors	30	120	>3600	>3600	288	340	315	81.4	**67.5**	250	194
3x10					39253	14576	10247	1038	788	35867	21465
Reactors2	10	24	>3600	>3600	28.8	**9.5**	12.3	10.4	12.2	9.93	11.9
2x5					128359	4908	10850	4344	5802	5597	5353
Sierp3Bis(eq)	83	6	>3600	>3600	4917	>3600	>3600	>3600	**389**	>3600	4503
1x14,6x6,15x2,3x1					44803				218		122409
Trigexp1	30	1	>3600	13	**0.08**	**0.08**	**0.08**	**0.08**	0.09	0.08	0.08
6x5				27	1	1	1	1	1	1	1
Trigexp2	11	0	1554	>3600	83.7	81.2	85.7	105	83.0	**80.6**	82.1
2x4,2x3			2116259		16687	15771	16755	3797	2379	15771	11795

Table 7.2: Results on structured benchmarks.

Table 7.2 reports the results on structured benchmarks. The same protocol as above has been followed, except that the solving strategy is more sophisticated. Between two bisections, the propagation with subsystems follows a 3BCID contraction and an interval Newton. The four S-kB columns report the results obtained by the S-kB-Revise variant. The results obtained by Box-k-Revise are generally worse and appear, with multisplit only, in the last two columns.

Standard strategies based on HC4 or Box followed by interval Newton are generally not competitive with Bok-k on the tested benchmarks. The solving strategy based on S-kB-Revise with subContractor=3BCID (column S-kB(3BCID)) appears to be a robust hybrid algorithm that is never far behind 3BCID and is

sometimes clearly better. The gain w.r.t. `3BCID` falls indeed between 0.7 and 12. The small number of boxes highlights the additional filtering power brought by well-constrained subsystems. Again, multisplit is often the best option.

The success of `Box-k` on `Sierp3Bis(eq)` has led us to try a particular version of `IBB` in which the inter-block filtering [Neveu et al., 2005] is performed by `3BCID`. Although this variant seldom shows a good performance, it can solve `Sierp3(eq)` in 330 seconds.

7.5.3 Benefits of sophisticated features

Tables 7.3 has finally been added to show the individual benefits brought by two features: the user parameter $\tau_{\rho_{io}}$ driving the procedure `ProcessLeaf?` and the backtrackable list of leaves used to reuse the job achieved inside the subsystems.

Table 7.3: Benefits of the backtrackable data structure (BT) and of $\tau_{\rho_{io}}$ in the `Box-k`-based strategy. Setting $\tau_{\rho_{io}} = \infty$ means that subsystem leaves will be always processed in the revise procedure.

	Chair	**Latham**	**Ponts(eq)**	**Ponts(geo)**	**Sierp3(geo)**	**Star**	**Tan(eq)**	**Tan(geo)**	**Tetra**
BT, $\tau_{\rho_{io}}$	0.52	1.08	0.71	0.31	1.38	0.12	0.08	0.19	0.76
¬ BT, $\tau_{\rho_{io}}$	10.8	4.61	1.51	1.27	23.9	2.34	0.71	1.58	2.13
BT, $\tau_{\rho_{io}} = \infty$	23.4	4.71	2.60	1.00	23.8	1.67	1.09	1.81	3.57
¬ BT, $\tau_{\rho_{io}} = \infty$	24.2	6.60	2.80	1.11	23.9	2.40	1.15	1.82	3.54
	Bratu	**Brent**	**BroyB.**	**BroyT.**	**Sierp3B(eq)**	**Reac.**	**Reac.2**	**Trigexp1**	**Trigexp2**
BT, $\tau_{\rho_{io}}$	33.0	20.2	0.15	0.24	389	67	12.2	0.08	83
¬ BT, $\tau_{\rho_{io}}$	33.2	21.0	0.14	0.23	411	97	12.0	0.07	85
BT, $\tau_{\rho_{io}} = \infty$	33.9	23.8	0.38	0.28	519	164	13.1	0.10	103
¬ BT, $\tau_{\rho_{io}} = \infty$	33.0	28.7	0.40	0.38	533	401	18.7	0.07	148

Every cell reports the best result (CPU time in second) among both subcontractors. Multisplit is allowed in all the tests. The first line of results corresponds to the implemented and sophisticated revise procedure; the next ones correspond to simpler versions for which at least one of the two advanced features has been removed.

Three main observations can be drawn. First, when a significant gain is brought by the features on a given system, then this system is efficiently handled against competitors in Tables 7.1 and 7.2. Second, $\tau_{\rho_{io}}$ seems to have a better impact on performance than the backtrackable list, but the difference is slight. Third, several systems are only slightly improved by one of both features, whereas the gain is significant when both are added together. This is true for most of the `IBB` benchmarks. On these systems it often occurs that a job inside *several* subsystems (during the same propagation process) leads to identify atomic boxes (some others are not fully explored thanks to $\tau_{\rho_{io}}$). Although we multisplit only one of these subsystems, the job on the others is saved in the backtrackable list.

7.6 Conclusion

We have proposed a new type of filtering algorithms handling $k \times k$ well-constrained subsystems in an NCSP. $k \times k$ interval Newton calls and selected bisections inside such subsystems are useful to better contract decomposed and structured NCSPs. In addition, the local trees built inside subsystems allow a solving strategy to learn interesting choice points splitting several variable domains simultaneously.

Solving strategies based on Box-k propagations and multisplit have mainly three parameters: the choice between Box-k-Revise and S-kB-Revise (although Box-k-Revise seems better suited only for decomposed systems), the choice of subcontractor (although 3BCID seems to be often a good choice), and $\tau\rho_{io}$. This last parameter appears to be finally the most important one.

On decomposed and structured systems, our first experiments suggest that our new solving strategies are more efficient than standard general-purpose strategies based on HC4, Box or 3BCID (with interval Newton). Box-k+multisplit can be viewed as a generalization of IBB. It can also solve large decomposed NCSPs with relatively small blocks in less than one second, but can also handle structured NCSPs that IBB cannot treat.

If a system is decomposable, then there exists a partial order between well-constrained subsystems, i.e., a DAG of blocks can be found. Thus, IBB can work with this DAG of blocks to compute the solutions. However, no such subsystems can be extracted if the system is not decomposable. Box-k can go further by not imposing any order between its subsystems. The blocks may form circuits. This is not an issue for a propagation algorithm that can work with a cyclic constraint graph. That is why Box-k can work with irreducible (while structured) systems. However, a crucial question remains: How to automatically determine subsystems provided to a Box-k based strategy?

We could imagine the following principle for selecting a promising subsystem. Consider the NCSP $P = (X, C, [B])$).

1. As a preprocessing, based on the material concerning the impact computation introduced in Section 7.3.4, we compute all the pairs (c, x) (with $c \in C$ and $x \in X$) such that the impact of the variable x on the constraint c is weak.
2. We remove the *weak* arcs corresponding to these pairs from the constraint graph.
3. We use maximum-matching machinery to extract a well-constrained subsystem of k constraints related to exactly k variables (see the last part of [Bliek et al., 1998]).

Finally, instead of partitioning the pairs into two parts (the *weak* one and the *strong* ones) and use a maximum matching, we could maybe take into account the real-valued impact with a flow-based algorithm.

Chapter 8

Conclusions and future perspectives

In this thesis, we have presented several contributions related to interval-based methods and solving of systems of equations.

The first of them, `Mohc` (see Chapter 4), is a constraint propagation algorithm. It uses several procedures exploiting the monotonicity of a function f in an adaptive way, i.e., only when the evaluation by monotonicity of f is better enough than the natural evaluation of f (identifying a condition for which the procedures exploiting the monotonicity deserve to be used). Experiments have shown that `Mohc` has a potential to advantageously replace the classical contractors `HC4` and `Box`.

In Section 4.2.1 we have presented the algorithm `MinMaxRevise` of `Mohc`. The procedure allows us, using the monotonicity of a function f, to contract:

- variables appearing once in f,
- occurrences of non-monotonic variables of f, and
- *some* occurrences of monotonic variables of f (thanks to the improvements proposed in Section 4.7).

`MinMaxRevise` performs all the contractions in time $O(e)$, where e is the number of unary and binary operations of the function. We are working on a new improvement of this procedure (`MinMaxRevise2`) that allows us to maximize the contraction of each occurrence of monotonic variables. It consists, basically, in performing a simple symbolic manipulation that transforms an occurrence x in the linear expression $ax'+(1-a)x$ (with $x'=x$). The value of a should be chosen such that the projection over x' is maximized and the monotonicity of the *projection function* related to x' is maintained.

Consider a simple constraint $c : x+2x = 0$ with domain $[x] = [-1, 1]$. The classical contractor `HC4-Revise` can converge to the solution $x = 0$ if it is called until reaching the fixpoint. Instead, `MinMaxRevise2` would replace the first occurrence of x by the expression $ax' + (1 - a)x$ (with $a = 3$), i.e., $c' : (3x' - 2x) + 2x = 0$. Then, the procedure would project over x' evaluating the projection function ($p_{x'}(x) = \frac{2x-2x}{3}$) by monotonicity, i.e.:

$$[x'] \leftarrow [x'] \cap [p_{x'}]_m([x]) = [0, 0]$$

In the example, the hull-consistency of c is enforced with only one call to `MinMaxRevise2`. We believe that this algorithm would have the same time complexity as `MinMaxRevise` and `HC4`, i.e., $O(e)$.

A second future work related to `Mohc` consists in allowing the `MonotonicBoxNarrow` procedure (described in Section 9) to treat non-monotonic variables (recall the procedure treats only monotonic variables), i.e.

the variables in W. In this case, the procedure should be closer to the classical `BoxNarrow` procedure (described in Section 3.2.3.5) but using the evaluation by monotonicity instead of the natural evaluation. The main motivation comes from the fact that if a given function f has only one variable w in W, then the classical `BoxNarrow` procedure, using the evaluation by monotonicity, is able to project *optimally* on w.

Another future work related to `Mohc` consists in implementing an adaptive version of the τ_{mohc} parameter (described in Section 4.3.1). Recall that this parameter allows us to decide whether the monotonic-based procedures are executed with a function f according to a ratio (ρ_{mohc}) between the diameter of the evaluation by monotonicity and the diameter of the natural evaluation of f. The experiment about the application frequency of the monotonicity-based procedures (see Section 4.6.1.4) suggests that when the ρ_{mohc} ratio of a function f has *often* high values (e.g., $\rho_{mohc}[f] > 0.7$) it is worth increasing τ_{mohc} for using more often the monotonic-based procedures in f. Otherwise, it is worth decreasing τ_{mohc}.

Our second contribution, *Occurrence Grouping* (described in Chapter 5) is a new method for computing the image of functions that improves the evaluation by monotonicity. The Occurrence Grouping method creates for each variable three auxiliary, respectively increasing, decreasing and non monotonic variables in f. Then it transforms f into a function f^{og} that groups the occurrences of a variable into these auxiliary variables. As a result, the *evaluation by occurrence grouping* of f, i.e., the evaluation by monotonicity of f^{og}, is better than the evaluation by monotonicity of f.

Three points suggest to combine constraints linearly such that the monotonicity of the combination brings an additional contraction:

- The possibility of computing the hull-consistency in monotonic functions with multiple occurrences of variables.
- The reduction of the dependency problem thanks to the occurrence grouping method.
- Our sophisticated algorithm `MinMaxRevise'` (soon `MinMaxRevise2`) that allows us to compute projections over each variable occurrence using the monotonicity.

The third contribution is related to the algorithm `I-CSE` (described in Chapter 6), a variant of the well-known Common Subexpression Elimination (CSE) technique. `I-CSE` is used as a preprocessing technique. It replaces the common subexpressions of a system by auxiliary variables for improving the filtering power of constraint propagation algorithms (in particular `HC4`). Experiments have shown significant gains of one or several orders of magnitude on several benchmarks when `I-CSE` is used.

In a short term, we see two interesting works related to `I-CSE`. The first one has been introduced briefly in Section 6.5. It consists in using techniques from symbolic computation for finding more CSs (e.g., factorizations using the Horner scheme). Consider the equations $c_1 : 2x+2y = a$ and $c_2 : yx+y^2+z = b$. At a first glance, they do not share CS. However, the transformation into $c_1 : 2(x + y) = a$ and $c_2 : y(x + y) + z = b$ makes appear the CS $x + y$.

The second future work consists in making constraint propagation algorithms compatible with the additional contraction provided by `I-CSE`. The replacement of subexpressions by auxiliary variables improves the contraction filtering by adding new constraints but it also increases the *locality problem* due to the decomposition of constraints (see Section 3.6). `HC4` is not impacted by this decomposition, because it already works with a decomposition of the system in primitive constraints. However, more sophisticated contractors like `Box` and `Mohc` may show a loss in performance.

Our last contribution, `Box-k` (described in Chapter 7), consists in a new type of filtering algorithms handling $k \times k$ well-constrained subsystems in an NCSP. $k \times k$ interval Newton calls and selected bisections inside such subsystems are useful to better contract decomposed and structured NCSPs. In addition, the local trees built inside subsystems allow a solving strategy to learn interesting choice points splitting several variable domains simultaneously.

Section 7 has described a way to automatically determine subsystems provided to a `Box-k` based strategy.

Appendix A

Proofs of Properties Related to Mohc

A.1 Proof of Lemma 4

First recall that in every node of the expression tree T_{fmax} (resp. T_{fmin}) representing f_{max} (resp. f_{min}), there is no overestimation due to the variables in X because they are replaced by points. The proof of Lemma 4 is based on Lemma 6 proved below and Lemma 2 (see Section 4.4.2, page 81) which shows that a second call to `MinMaxRevise` following the call to `MonotonicBoxNarrow` would not bring additional contraction to any $[y_j]$.

Lemma 6 *Two traversals of T_{fmax} (with `HC4-Revise`) are sufficient to reach a fixpoint in contraction on variables $y_j \in Y$ occurring once in T_{fmax}.*

Proof 15 *Since f is a continuous function and, for every $y_j \in Y$, $0 \notin \frac{\partial f}{\partial y_j}([B])$, then the (bottom-up) evaluation or (top-down) narrowing functions g called in every node during `HC4-Revise` of T_{fmin} (or T_{fmax}) are continuous and monotonic in every of their arguments intervals. This comes from the rule of composition of functions stating $\frac{\partial f}{\partial y_j}([B]) = \frac{\partial f}{\partial g}(g([B])) \times \frac{\partial g}{\partial y_j}([B])$. Since the multiplication preserves the 0 in the resulting interval, $0 \notin \frac{\partial f}{\partial y_j}([B])$ implies that $0 \notin \frac{\partial f}{\partial g}(g([B]))$ and $0 \notin \frac{\partial g}{\partial y_j}([B])$. The same rule applies to two nodes g_1 and g_2 for which g_2 is an argument of g_1: $\frac{\partial f}{\partial g_2}([B]) = \frac{\partial f}{\partial g_1}(g_1([B])) \times \frac{\partial g_1}{\partial g_2}([B])$. The same reasoning yields that $0 \notin \frac{\partial g_1}{\partial g_2}([B])$. Thus, every bottom-up evaluation function g is monotonic in every of their arguments intervals. The same reasoning applies to the top-down narrowing functions g since an "inverse" function of a monotonic (continuous) function is also monotonic.*

Because every node function g in T_{fmin} (or T_{fmax}) is monotonic, g computes an arc-consistent output interval $[z_g]$, i.e., every real number $z \in [z_g]$ has a support in the input intervals. Since T_{fmin} (or T_{fmax}) is a tree, the two traversals of T_{fmin} performed by `HC4-Revise` then optimally narrow every $y_j \in Y$. This is an application [Faltings, 1994] of a result by Freuder [Freuder, 1982] concerning the finite-domain CSPs stating that two (directed) arc-consistent traversals of an acyclic CSP makes it globally-consistent. That is, no propagation loop is necessary and the two traversals are sufficient to reach the fixpoint in filtering. □

A.2 Proof of Lemma 5, page 84

First recall that MonotonicBoxNarrow cannot result in an empty box (no solution) because MinMaxRevise has been called before with success (see Lemma 1). This precondition implies that for any bound of each $x_i \in X$, only the three cases depicted in Figure 4.3, page 82, may occur. When the case 1 occurs, the dichotomic narrowing process performed by LeftNarrowFmax (or symmetric procedures) converges onto a final interval $[l]$ including surely a zero of $f_{max}^{x_i}$. Contrarily to the classical LeftNarrow procedure used by Box [Benhamou et al., 1999; Van Hentenryck et al., 1997], $[l]$ surely contains the zero because the $f_{max}^{x_i}$ evaluations lead to no overestimation (see proof of Lemma 4). When the cases 2 or 3 occur (see Figure 4.3), $[x_i]$ cannot be (left) narrowed because the bound is a zero of the function.

After only one loop on each variable $x_i \in X$ ($i \in 1, ..., n$), each of the $2 \times n$ bounds of $[x_i]$ are optimal either because computed by the dichotomic process (case 1 explained above), or because applyFmin([i])=true (or applyFmax([i])=true), which ensures that no further reduction is expected in these cases 2 or 3 (see Section 4.4.3). □

A.3 Proof of Proposition 10, page 84

Proposition 10 follows Lemmas 4 and 5. One call to MinMaxRevise and one loop on the monotonic variables in X ensure that hull-consistency is achieved. □

A.4 Proof of Proposition 11, page 84

Lemma 5 also holds for Proposition 11, but Lemma 7 must replace Lemma 4.

Lemma 7 *With the same hypotheses as in Proposition 11, one call to* MinMaxRevise' *(calling* TAC-revise*) contracts optimally every* $[y_j] \in [Y]$.

Lemma 7 can be proved using the proof of Lemma 6 and the correctness of TAC-revise.

Remark. We assume in Proposition 11 that f is continuous in the current box. This hypothesis can be relaxed provided that the bottom-up expression evaluations are performed by a "TAC-eval" procedure that would combinatorially combine the continuous parts extracted in the different nodes of the expression tree.

A.5 Proof of Proposition 12 (time complexity), page 85

Calls to HC4-Revise (two traversals of an expression tree), GradientCalculation (computing all the partial derivatives also amounts in two tree traversals) and MinMaxRevise are $O(e)$. One call to OccurrenceGrouping is $O(n\,k\,log_2(k))$. The complexity of MonotonicBoxNarrow is time $O(n\,e\,log_2(\frac{1}{\epsilon}))$ (see page ??). One Newton iteration takes $O(e)$ (one function evaluation, plus one gradient calculation). The maximum number of possible slices in one interval $[x_i]$ is $\frac{1}{\epsilon}$. The number of Newton iterations is $O(log_2(\frac{1}{\epsilon}))$ (see Lemma 9). In LazyMohc-Revise, MonotonicBoxNarrow is time $O(n)$ because are called only two constant-time Newton iterations per interval.

Overall, the time complexity of Mohc-Revise is $O(e)$ plus $O(n\,k\,log_2(k))$ (if OccurrenceGrouping is called), plus $O(n\,e\log(\frac{1}{\epsilon}))$. The time complexity of LazyMohc-Revise is $O(e)$ plus $O(n\,k\,log_2(k))$ (if OccurrenceGrouping is called).

□

A.6 The LazyMonotonicBoxNarrow procedure

Algorithm 23 LazyMonotonicBoxNarrow (**in-out** $[B]$; **in** f_{max}, f_{min}, X, $[G]$, ϵ)

```
1:  for all variable x_i ∈ X do
2:    z_max ← overline([f_max]([B])); [z_min] ← underline([f_min]([B]))
3:    if z_max > 0 and applyFmax[i] then
4:      if [g_i] > 0 then
5:        [l] ← [x_i] ∩ (overline(x_i) − z_max/[g])
6:        if underline(l) > underline(x_i) /* x_i is contracted */ then
7:          ∀j ≠ i : applyFmin[i] ← false
8:        end if
9:      else
10:       [r] ← [x_i] ∩ (underline(x_i) − z_max/[g])
11:       if r̄ < overline(x_i) then ∀j ≠ i : applyFmin[i] ← false  end if
12:     end if
13:   end if
14:   if z_min < 0 and applyFmin[i] then
15:     if [g_i] > 0 then
16:       [r] ← [x_i] ∩ (underline(x_i) − z_min/[g])
17:       if r̄ < overline(x_i) then ∀j ≠ i : applyFmax[i] ← false  end if
18:     else
19:       [l] ← [x_i] ∩ (overline(x_i) − z_min/[g])
20:       if l̲ > underline(x_i) then ∀j ≠ i : applyFmax[i] ← false  end if
21:     end if
22:   end if
23:   [x_i] ← [l̲, r̄]
24: end for
```

Algorithm 23 is a simplified version of the MonotonicBoxNarrow procedure (see Algorithm 9). It only calls the first and cheap Newton iteration described in Section *Lazy evaluations of f_{min} and f_{max}*, page 83, for every bound of x_i in X. Lines 7, 11, 17 and 20 corresponds to the improvements related to arrays applyFmin and applyFmin explained in Section 4.4.4.

A.7 The latest version of Mohc-Revise algorithm

Algorithm 24 is the optimized version of Mohc-Revise (and LazyMohc-Revise). Observe that the procedure calls first LazyMonotonicBoxNarrow and then MonotonicBoxNarrow. As LazyMonotonicBoxNarrow

Algorithm 24 Mohc-Revise (**in-out** $[B]$; **in** f, Y, W, ρ_{mohc}, τ_{mohc}, ϵ)

```
HC4-Revise(f(Y,W) = 0, Y, W, [B])
if W ≠ ∅ and ρ_mohc[f] ≤ τ_mohc then
  ([G], [G_o]) ← GradientCalculation(f, W, [B])
  (f^og, W) ← OccurrenceGrouping(f, W, [B], [G_o])
  (f_max, f_min, X, W) ← ExtractMonotonicVars(f^og, W, [B], [G])
  MinMaxRevise([B], f_max, f_min, Y, W)
  LazyMonotonicBoxNarrow([B], f_max, f_min, X, [G], ϵ)
  /* Comment the next line in LazyMohc-Revise */
  MonotonicBoxNarrow([B], f_max, f_min, X, [G], ϵ)
end if
```

is cheap, calling first allows us to set to *false* some values of the `applyFmin` and `applyFmax` arrays before calling the dichotomic and more expensive `MonotonicBoxNarrow` procedure (Algorithm 9, page 83).

A.8 The `LeftNarrowFmax` procedure revisited

The fact that our `LeftNarrowFmax` procedure works with the punctual function $\overline{[f^x_{max}]}$ using floating point precision might imply the loss of the unique solution due to floating point errors. When this happens a modification of `LeftNarrowFmax` allows the algorithm to be conservative returning a lower bound of L.

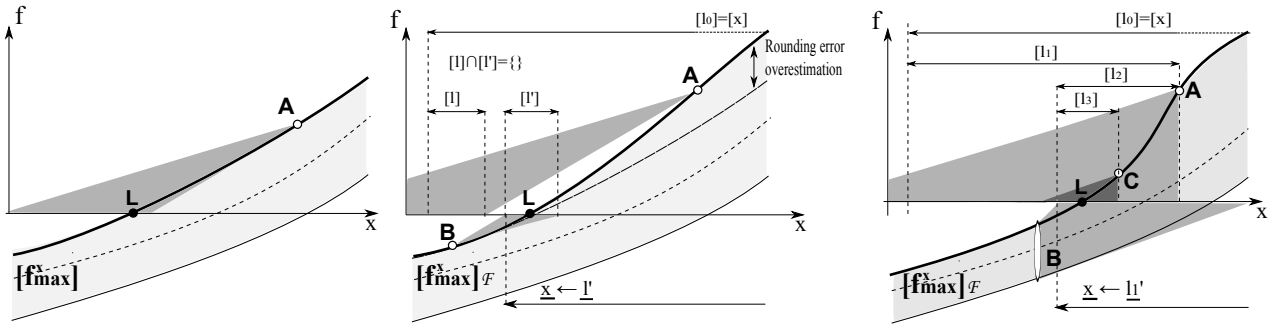

Figure A.1: (**Left**) Newton iteration when $[f^x_{max}]$ is computed with real precision. (**Center**) Newton iteration returns an empty box due to rounding errors. (**Right**) A Newton variant (Algorithm 26) converging to L in log-time.

In Figure A.1-left we can see the ideal case. $[f^x_{max}]$ is evaluated over the *real* numbers, i.e., without floating point errors. In this case, the Newton iteration *always* projects over a segment in $[x]$ containing the solution (projection of the point A). Figure A.1-center shows $[f^x_{max}]_{\mathbb{F}}$ evaluated using floating point precision[1]. As the interval derivative used by the Newton operator is related to $[f^x_{max}]$ (a real interval function), it can project over a segment that does not contain the solution. In this case the univariate Newton method could return an empty interval (projections of A and B have no intersection).

We propose two solutions:

1. (Algorithm 25) We know that, after performing successfully the test of existence (line 3), there exists only one solution in $[x]$. Then, when the case of Figure A.1-center happens (i.e., $[l] \cap [l'] = \emptyset$),

[1]The rounding errors are generally of few u.l.p.s.

the algorithm contracts $[x]$ using the *rightmost left bound* between the current interval ($[l]$) and the next Newton projection over x ($[l']$) (i.e., $\underline{x} \leftarrow max(\underline{l}, \underline{l'})$). In the figure: $[x] \leftarrow [\underline{l'}, \overline{x}]$. This procedure allows a fast convergence of the Newton method but the distance between L and the new left bound of $[x]$ could be theoretically a bit greater than the required precision.

2. (Algorithm 26) The second solution produces always an interval sharper than the precision and maintains the logarithmic convergence (it is illustrated in Figure A.1-right). The Newton operator is replaced by the classic Newton operator (using all the evaluation interval), i.e.,

$$[l] \leftarrow x_m - \frac{[f^x_{max}](x_m)}{[g]} \quad \text{(instead of } \; [l] \leftarrow x_m - \frac{\overline{[f^x_{max}](x_m)}}{[g]}\text{)}$$

Thus, if $\overline{[f^x_{max}](x_m)}$ is positive (meaning the solution is in the right side of x_m): $[l] \leftarrow [l] \cap [-\infty, x_m]$ (line 10). Otherwise, if $\overline{[f^x_{max}](x_m)}$ is negative, then the intersection is not necessary because Newton considers automatically the left side of x_m (the projection of B in Figure A.1-right).

Algorithm 25 `LeftNarrowFmax`$_1$ (**in-out** $[x]$; **in** f^x_{max}, $[g]$, ϵ, i)

```
1:  [l] ← [x]
2:  [z_l] ← [f^x_max](x_)
3:  if z̄_l < 0 /* test of existence */ then
4:     [l] ← [l] ∩ ( x_ − z̄_l/[g] ) /* second cheap Newton iteration */
5:     size ← ε × Diam([l])
6:     while [l] ≠ ∅ and Diam([l]) > size do
7:        x_m ← Mid([l]); z_m ← overline([f^x_max](x_m))   /* z_m ← underline([f^x_min](x_m)) in {Left|Right}NarrowFmin */
8:        [l'] ← x_m − z_m/[g]
9:        if [l] ∩ [l'] ≠ ∅ then
10:          [l] ← [l] ∩ [l']   /* Newton iteration */
11:       else
12:          ∀j ≠ i : applyFmax[i] ← false
13:          [x] ← [max(l_, l'_), x̄]; return
14:       end if
15:    end while
16:    if l_ > x_ then ∀j ≠ i : applyFmax[i] ← false  end if
17:    [x] ← [l_, x̄]
18: end if
```

Algorithms 25 and 26 are sophisticated versions of the `LeftNarrowFmax` procedure. Note that a cheap Newton iteration has been added after the existence test (the improvement is explained in Section 4.4.4). Also the updates to the `ApplyFmax` array have been added as detailed in Section 4.4.3.

We have implemented the first version (just because the idea arrived first). We still have to compare the two versions both in time and contraction power.

The other three procedures (`RightNarrowFmin`, `LeftNarrowFmin` and `RightNarrowFmax`) are also accordingly modified.

Algorithm 26 LeftNarrowFmax$_2$ (**in-out** $[x]$; **in** f^x_{max}, $[g]$, ϵ, i)

$[l] \leftarrow [x]$
$[z_l] \leftarrow [f^x_{max}](\underline{x})$
if $\overline{z_l} < 0$ /* test of existence */ **then**
  $[l] \leftarrow [l] \cap \left(x_m - \frac{[z_l]}{[g]}\right)$ /* second cheap Newton iteration */
  $size \leftarrow \epsilon \times \texttt{Diam}([l])$
  while $[l] \neq \emptyset$ **and** $\texttt{Diam}([l]) > size$ **do**
    $x_m \leftarrow \texttt{Mid}([l])$; $[z_m] \leftarrow [f^x_{max}](x_m)$ /* $[z_m] \leftarrow [f^x_{min}](x_m)$ in {Left|Right}NarrowFmin */
    $[l] \leftarrow [l] \cap \left(x_m - \frac{[z_m]}{[g]}\right)$
    if $\overline{z_m} > 0$ **then**
      $[l] \leftarrow [l] \cap [-\infty, x_m]$
    end if
  end while
  if $\underline{l} > \underline{x}$ **then** $\forall j \neq i$: applyFmax$[i] \leftarrow$ **false** **end if**
  $[x] \leftarrow [\underline{l}, \overline{x}]$
end if

Appendix B

Proofs of Properties Related to Occurrence Grouping

B.1 Proof of Proposition 16, page 108

Proof 16 *Due to propositions 28 and 29, there exist real values $r^*_{a_i}$, $r^*_{b_i}$ and $r^*_{c_i}$ inside the intervals computed by Algorithm11, such that the monotonicity conditions w.r.t. variables x_a and x_b are satisfied.*

Let us call $f^{og*}(x_a, x_b, x_c)$ the function $f(x)$ in which every occurrence x_i is replaced by the convex linear combination $r^*_{a_i}x_a + r^*_{b_i}x_b + r^*_{c_i}x_c$. The evaluation by monotonicity of f^{og*} over intervals $[x_a] = [x_b] = [x_c] = [x]$ is thus an overestimate of the hull of the image by f of all $x \in [x]$. The evaluation by monotonicity of the actual function f^{og} computed by Algorithm 11 using intervals $[r_{a_i}]$, $[r_{b_i}]$ and $[r_{c_i}]$ is an overestimate of the evaluation of f^{og*} and thus is also an overestimate of the hull of the image by f of all $x \in [x]$. □

Proposition 28 *Algorithm 12 (*`OG_case2`*) computes, for every occurrence i, intervals $[r_{a_i}]$, $[r_{b_i}]$ and $[r_{c_i}]$ that satisfy the constraints:*

$$(\forall i = 1 \dots k)(\exists r_{a_i} \in [r_{a_i}])(\exists r_{b_i} \in [r_{b_i}])(\exists r_{c_i} \in [r_{c_i}]) : \quad \begin{array}{c} r_{a_i} + r_{b_i} + r_{c_i} = 1 \\ \sum_{i=1}^{k} \underline{g_i} r_{a_i} \geq 0 \\ \sum_{i=1}^{k} \overline{g_i} r_{b_i} \leq 0 \end{array}$$

Proof 17 *For every occurrence i, according to conditions met by $[g_i]$, for r_{a_i} we choose the value $1 - \alpha_1$ (line 8), α_2 (line 10) or 0 (line 12); for r_{b_i}, we choose α_1, $1 - \alpha_2$ or 0; for r_{c_i}, we choose 0 or 1 (that are both reals and floats); where α_1 and α_2 are the following two real numbers (not necessarily floating point numbers):*

$$\alpha_1 = \frac{\underline{G^+}\overline{G^-} + \overline{G^-}\underline{G^-}}{\underline{G^+}\overline{G^-} - \underline{G^-}\overline{G^+}} \qquad \textit{and} \qquad \alpha_2 = \frac{\underline{G^+}\overline{G^+} + \overline{G^-}\underline{G^+}}{\underline{G^+}\overline{G^-} - \underline{G^-}\overline{G^+}}$$

First, thanks to the conservative interval-based computation of $[\alpha_1]$ and $[\alpha_2]$ performed at lines 3 and 4, we have $\alpha_1 \in [\alpha_1]$ and $\alpha_2 \in [\alpha_2]$.

Second, this is straightforward to check at lines 8, 10 and 12 that the chosen values for r_{a_i}, r_{b_i}, r_{c_i} *verify the relation* $r_{a_i} + r_{b_i} + r_{c_i} = 1$.

Finally, the main difficulty consists in proving that, for any occurrence i, $\sum_{i=1}^{k} \underline{g_i} r_{a_i} \geq 0$ *and* $\sum_{i=1}^{k} \overline{g_i} r_{b_i} \leq 0$. *First,* α_1 *and* α_2 *are computed by analytically solving the equations:*

$$\begin{array}{rcl} (1-\alpha_1) \times \underline{G^+} + \alpha_2 \times \underline{G^-} & = & 0 \\ \alpha_1 \times \overline{G^+} + (1-\alpha_2) \times \overline{G^-} & = & 0 \end{array}$$

The values $\underline{G^+}$, $\overline{G^+}$, $\underline{G^-}$ *and* $\overline{G^-}$ *are computed in lines 1 and 2 of the algorithm. Due to floating point errors, these values are overestimated, i.e.,* $\underline{G^+} \leq \sum_{[g_i] \geq 0} \underline{g_i}$, $\overline{G^+} \geq \sum_{[g_i] \geq 0} \overline{g_i}$, $\underline{G^-} \leq \sum_{[g_i] \leq 0} \underline{g_i}$ *and* $\overline{G^-} \geq \sum_{[g_i] \leq 0} \overline{g_i}$.

Thus, we obtain:

$$\begin{array}{rcl} \sum_{i=1}^{k} \underline{g_i} r_{a_i} = (1-\alpha_1) \sum_{[g_i] \geq 0} \underline{g_i} + \alpha_2 \sum_{[g_i] \leq 0} \underline{g_i} & \geq & (1-\alpha_1) \times \underline{G^+} + \alpha_2 \times \underline{G^-} = 0 \\ \sum_{i=1}^{k} \overline{g_i} r_{b_i} = \alpha_1 \sum_{[g_i] \geq 0} \overline{g_i} + (1-\alpha_2) \sum_{[g_i] \leq 0} \overline{g_i} & \leq & \alpha_1 \times \overline{G^+} + (1-\alpha_2) \times \overline{G^-} = 0 \end{array}$$

□

Proposition 29 *Algorithm 13 (*`OG_case3+`*), for every occurrence* i, *computes intervals* $[r_{a_i}]$, $[r_{b_i}]$ *and* $[r_{c_i}]$ *that satisfy the constraints:*

$$(\forall i = 1 \ldots k)(\exists r_{a_i} \in [r_{a_i}])(\exists r_{b_i} \in [r_{b_i}])(\exists r_{c_i} \in [r_{c_i}]) : \quad \begin{array}{c} r_{a_i} + r_{b_i} + r_{c_i} = 1 \\ \sum_{i=1}^{k} \overline{g_i} r_{b_i} \leq 0 \\ \sum_{i=1}^{k} \underline{g_i} r_{a_i} \geq 0 \end{array}$$

Proof 18 *Let us denote by* $i' = index(j')$ *the occurrence* i *studied at lines 15, 16 of the algorithm. We distinguish three main cases:* $i < i'$ *(lines 2–13),* $i > i'$ *(lines 18–21) and* $i = i'$ *(lines 15–16):*

- $i < i'$*: We choose* $(r_{a_i}, r_{b_i}, r_{c_i}) = (1, 0, 0)$.
  (Recall that 0 and 1 are floating point numbers.)
- $i > i'$ *(*$i \in \{index(j'+1), \ldots, index(k)\}$*):*
  We choose $(r_{a_i}, r_{b_i}, r_{c_i}) = (0, 0, 1)$.
- $i = i'$*: We choose* $(r_{a'_i}, r_{b'_i}, r_{c'_i}) = (\underline{\alpha}, 0, 1 - \underline{\alpha})$. *That is, for* $r_{a'_i}$, *we select the lower bound* $\underline{\alpha}$ *of the interval* $[\alpha]$ *computed in line 15 of the algorithm.*

First, this is straightforward to check in the three cases that the chosen values for r_{a_i}, r_{b_i}, r_{c_i} *verify the relation* $r_{a_i} + r_{b_i} + r_{c_i} = 1$.

Second, this is also straightforward to check $\sum_{i=1}^{k} \overline{g_i} r_{b_i} \leq 0$ *since, for all* i*, we always select the value 0 for* r_{b_i}.

Finally, the main difficulty consists in proving that, for any occurrence i, $\sum_{i=1}^{k} \underline{g_i} r_{a_i} \geq 0$.

We have: $\sum_{i=1}^{k} \underline{g_i} r_{a_i} = \sum_{[g_i] \geq 0} \underline{g_i} + \sum_{\substack{i=index(j) \\ j=1}}^{j=j'-1} \underline{g_i} + \underline{g_{i'}} r_{a_{i'}}$.

Let $\underline{\dot{g}_a}$ *denote the real number* $\sum_{[g_i] \geq 0} \underline{g_i} + \sum_{\substack{i=index(j) \\ j=1}}^{j=j'-1} \underline{g_i}$ *(*$\underline{\dot{g}_a}$ *is not necessarily a floating point number.) By construction, we have* $\underline{\dot{g}_a} + \underline{g_{i'}} \alpha = 0$, *where* $\alpha = -\frac{\underline{\dot{g}_a}}{\underline{g_{i'}}}$. *Indeed,* $\underline{\dot{g}_a} - \underline{g_{i'}} \frac{\underline{\dot{g}_a}}{\underline{g_{i'}}} = 0$.

However, the real number α *does not necessarily belong to the interval* $[\alpha] = \frac{\underline{g_a}}{\underline{g_{i'}}}$ *computed at line 15 (because of the overestimate induced by rounding errors in the sum* $\sum_{[g_i] \geq 0} \underline{g_i} + \sum_{\substack{i=index(j) \\ j=1}}^{j=j'-1} \underline{g_i}$ *over the floats).*

Fortunately, we have $0 \leq \underline{g_a} \leq \underline{\dot{g}_a} = \sum_{[g_i] \geq 0} \underline{g_i} + \sum_{\substack{i=index(j) \\ j=1}}^{j=j'-1} \underline{g_i}$ *and thus* $\underline{\alpha} \leq \alpha$.

Finally, recalling that $g_{i'} \leq 0$, *we obtain:*

$$\sum_{i=1}^{k} \underline{g_i} r_{a_i} = \underline{\dot{g}_a} + \underline{g_{i'}} \underline{\alpha} \geq \underline{\dot{g}_a} + \underline{g_{i'}} \alpha = 0$$

□

B.2 Proof of Proposition 17, page 108

Proof 19 *(***Proposition 17***) Following Lemma 8, we can rewrite the objective function as* $G = \texttt{Diam}([G_0]) + G_1 + G_2$, *where* $G_1 = \underline{g_a} - \overline{g_b}$ *and* $G_2 = \sum_{i=1}^{k} \big((|[g_i]| - \texttt{Diam}([g_i]) r_{c_i}) \big)$. *The term* $\texttt{Diam}([G_0]([B]))$ *is constant; Lemmas 9 and 10 show that the algorithm finds values* r_{a_i}, r_{b_i} *and* r_{c_i} *that minimize* G_1 *and* G_2 *resp. Thus, Algorithm 12 finds an optimal solution for* G. □

Lemma 8 *The objective function (5.3) can be rewritten as:*

$$G = \texttt{Diam}([G_0]) + \underline{g_a} - \overline{g_b} - \texttt{Diam}([g_c]) + \sum_{i=1}^{k} \big(|[g_i]| r_{c_i} \big) \tag{B.1}$$

where $[G_0] = \sum_{i=1}^{k} [g_i]$.

Proof 20 *(***Lemma 8***) We observe that* $[G_0] = [g_a] + [g_b] + [g_c]$ *and using interval arithmetic properties* $\texttt{Diam}([G_0]) = \texttt{Diam}([g_a]) + \texttt{Diam}([g_b]) + \texttt{Diam}([g_c])$.

Taking into account the constraints (5.4) and (5.5), page 102: $\texttt{Diam}([g_a]) = \overline{g_a} - \underline{g_a}$ *and* $\texttt{Diam}([g_b]) = -\underline{g_b} + \overline{g_b}$*. The diameter of* $[G_0]$ *can be written:* $\texttt{Diam}([G_0]) = \overline{g_a} - \underline{g_b} - (\underline{g_a} - \overline{g_b}) + \texttt{Diam}([G_c])$*. Replacing* $\overline{g_a} - \underline{g_b}$ *in (5.3) by* $\texttt{Diam}([G_0]) + \underline{g_a} - \overline{g_b} + \texttt{Diam}([G_c])$ *we obtain B.1.*

Lemma 9 *Let* P_1 *be the following linear program: find values* r_{a_i}, r_{b_i}, *and* r_{c_i} *for all* $i = 1..k$ *that minimize* $G_1 = \underline{g_a} - \overline{g_b}$ *subject to (5.4), (5.5),(5.6) and (5.7) (see page 102). If* $0 \in [G_m]$, *then Algorithm 12 finds an optimal of* P_1*.*

Proof 21 *Due to constraints (5.4) and (5.5), we deduce that* $G_1 \geq 0$*.*

If we analyze the `for` *loop (lines 6 to 14) of Algorithm 12 we note that each variable* r_{a_i} *gets the value* $1 - \alpha_1$ *if* $[g_i] \geq 0$*, gets the value* α_2 *if* $[g_i] < 0$*;* r_{a_i} *gets the value 0 otherwise. Then, at the end of the* `for` *loop we can compute:*

$$\underline{g_a} = (1 - \alpha_1) \sum_{[g_i] \geq 0} \underline{g_i} + \alpha_2 \sum_{[g_i] < 0} \underline{g_i} \tag{B.2}$$

As $\underline{G^+} = \sum_{[g_i] \geq 0} \underline{g_i}$ *and* $\underline{G^-} = \sum_{[g_i] < 0} \underline{g_i}$ *(lines 1-2 of the algorithm), Equation B.2 is reduced to:* $\underline{g_a} = (1 - \alpha_1)\underline{G^+} + \alpha_2 \underline{G^-}$*. Then replacing* α_1 *and* α_2 *by their values in lines 3-4 of the algorithm we obtain* $\underline{g_a} = 0$ *(in the same way we obtain* $\overline{g_b} = 0$*) respecting constraints (5.4) and (5.5) and finding the minimum of* $G_1 = 0$*. Constraint (5.6) remains satisfiable in lines 8, 10 and 12 of the algorithm.*

The satisfaction of Constraint (5.7) requires $0 \leq \alpha_1 \leq 1$ *and* $0 \leq \alpha_2 \leq 1$*. This is achieved in lines 3-4 and can be proved using some inequality properties of the parameters:* $\overline{G^+} \geq \underline{G^+}$ *and* $\overline{G^-} \geq \underline{G^-}$ *(inherent property of intervals);* $\underline{G^+} \geq 0$*, and* $\overline{G^-} \leq 0$*;* $-\overline{G^-} \leq \overline{G^+}$ *and* $\underline{G^+} \leq -\underline{G^-}$ *(deduced by the condition* $0 \in [G_m]([B])$ *where* $[G_m]([B]) = [G^+] + [G^-]$*).* □

Lemma 10 *Let* P_2 *be the linear program: find values* r_{a_i}, r_{b_i}, *and* r_{c_i} *for all* $i = 1..k$ *that minimize* $G_2 = \sum_{i=1}^{k} ((|[g_i]| - \texttt{Diam}([g_i]))\, r_{c_i})$*, subject to (5.6) and (5.7). Algorithm 12 finds an optimal solution of* P_2*.*

Proof 22 $|[g_i]| - \texttt{Diam}([g_i])$ *is less than 0 iff* $0 \in g[i]$*, then to minimize* G_2 *it is enough that for each* $g[i]$ *containing 0,* $r_{c_i} = 1$*. For respecting constraints (5.6) and (5.7) we add* $r_{a_i} = 0$ *and* $r_{b_i} = 0$*. This setting is achieved in line 12 of Algorithm 12.* □

B.3 Proof of Proposition 18, page 108

Definition 25 (Fractional knapsack problem) *Given a knapsack with capacity* M *and a list of* k *elements with weights* $W = \{w_1, ..., w_k\}$ *and values* $V = \{v_1, ..., v_k\}$*. The fractional knapsack problem* $FKP(M, W, V)$ *consists in finding the set* $R = \{r_1, ..., r_n\}$ *that maximizes the total value* $\sum_{i=1}^{k} v_i r_i$ *subject to* $M - \sum_{i=1}^{k} w_i r_i \geq 0$ *and* $r_i \in [0, 1]$*.*

Proof 23 *According to Lemma 11 the minimum of* G *can be found only if* $\forall i : \; r_{b_i} = 0$*. Then, following Lemma 12 the problem can be transformed into the fractional knapsack problem instance:* $FKP(M, \{w_1, ..., w_k\}, \{v_1, ..., v_k\})$*, where* $v_i = -(\overline{g_i} - |[g_i]|)$*,* $w_i = -\underline{g_i}$ *and* $M = \sum_{i=1}^{k_1} \underline{g_i}$*.*

Finally, proposition 30 shows that a greedy algorithm that put the items sorted by $\frac{v_i}{w_i}$ is capable of reaching the optimal solution. Algorithm 13 does exactly that. □

Proposition 30 *Let $FKP(M, W, V)$ an instance of the fractional knapsack problem. W.l.o.g., assume that W and P are sorted in decreasing order by the ratio $\frac{v_i}{w_i}$. It means that the given sets are sorted such that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq ... \geq \frac{v_k}{w_k}$. The optimal solution is found when:*

$$\begin{aligned} r_i &= 1 \qquad \text{for } \; i = 1, ..., s-1 \\ r_s &= \frac{M - \sum_{i=1}^{s-1} w_i}{w_s} \\ r_j &= 0 \qquad \text{for } \; j = s+1, ..., k \end{aligned}$$

In other words, we add elements to the solution set in order until we reach the capacity M or all the elements are used. Observe that there are at most one fractionary element in the solution set (r_s). All others are either full elements ($r_i = 1$), or are not part of the solution ($r_j = 0$).

Proof 24 *The proof can be found in [Martello and Toth, 1990].*

Lemma 11 *Let G be the objective function:*

$$G = \overline{g_a} - \underline{g_b} + \sum_{i=1}^{k} (|[g_i]|r_{c_i}) \tag{B.3}$$

subject to the constraints (5.4), (5.5),(5.6) and (5.7) (see page 102). If $\underline{G_m} \geq 0$ (where $[G_m]$ is the sum of the partial derivatives of the monotonic occurrences, i.e., $[G_m] = \sum_{i=1..k, 0 \notin [g_i]} [g_i]$), then for all grouping og_1 with $\sum_{i=0}^{k} r_{b_i} \neq 0$ (i.e., the set of occurrences x_b is not empty) there exists a grouping og_2, such that $G(og_2) \leq G(og_1)$.

Proof 25 *Consider a grouping og_1 such that the partial derivatives w.r.t. x_a, x_b and x_c are given resp. by:*

$$\begin{aligned} [g_a^1] &= [g_a^m] + [g_a^0] \\ [g_b^1] &= [g_b^m] + [g_b^0] \\ [g_c^1] &= [g_c^m] + [g_c^0] \end{aligned}$$

where $[g_a^m]$ (resp. $[g_b^m]$ and $[g_c^m]$) corresponds to the sum of the partial derivatives of the monotonic occurrences of x_a (resp. x_b and x_c), i.e., $[g_a^m] = \sum_{i=1..k, 0 \notin [g_i]} ([g_i] r_{a_i})$. $[g_a^0]$ (resp. $[g_b^0]$ and $[g_c^0]$) corresponds to the sum of the partial derivatives of the nonmonotonic occurrences of x_a (resp. x_b and x_c), i.e., $[g_a^0] = \sum_{i=1..k, 0 \in [g_i]} ([g_i] r_{a_i})$. We can compute $G(og_1)$ using (B.3):

$$G(og_1) = \overline{g_a^m} + \overline{g_a^0} - (\underline{g_b^m} + \underline{g_b^0}) + \sum_{i=1..k, 0 \notin [g_i]} (|[g_i]|r_{c_i}) + \gamma$$

where

$$\gamma = \sum_{i=1..k, 0 \in [g_i]} (|[g_i]| r_{c_i})$$

Consider a grouping og_2 such that the partial derivatives w.r.t. x_a, x_b and x_c are given resp. by:

$$\begin{array}{rcl} [g_a^2] & = & [g_a^m] + [g_b^m] + [g_c^m] + \alpha \times [g_a^0] \\ [g_b^2] & = & [0, 0] \\ [g_c^2] & = & [g_c^0] + (1-\alpha) \times [g_a^0] + [g_b^0] \end{array}$$

where $0 \leq \alpha \leq 1$. To create the grouping og_2 from og_1 is always possible. A way consists in 'moving' all the monotonic occurrences, from x_b and x_c, to x_a. Maybe, we would also need to move a fraction $(1-\alpha)$ of each non monotonic occurrence from x_a to x_c for keeping satisfied the constraint (5.4), i.e. $\underline{g_a^1} \geq 0$.

We can compute $G(og_2)$:

$$G(og_2) = \overline{g_a^m} + \overline{g_b^m} + \overline{g_c^m} + \alpha \times \overline{g_a^0} + \gamma + \beta$$

where

$$\beta = (1-\alpha) \sum_{i=1..k, 0 \in [g_i]} (|[g_i]| r_{a_i}) + \sum_{i=1..k, 0 \in [g_i]} (|[g_i]| r_{b_i})$$

As $0 \in [g_i]$ implies that $|[g_i]| \leq (\overline{g_i} - \underline{g_i})$:

$$\beta \leq (1-\alpha) \sum_{i=1..k, 0 \in [g_i]} \big((\overline{g_i} - \underline{g_i}) r_{a_i}\big) + \sum_{i=1..k, 0 \in [g_i]} \big((\overline{g_i} - \underline{g_i}) r_{b_i}\big)$$

Finally,

$$\beta \leq (1-\alpha)(\overline{g_a^0} - \underline{g_a^0}) + (\overline{g_b^0} - \underline{g_b^0})$$

Now we can compute the subtraction $G(og_1) - G(og_2)$:

$$\begin{array}{rcl} G(og_1) - G(og_2) & = & \overline{g_a^m} + \overline{g_a^0} - \underline{g_b^m} - \underline{g_b^0} + \sum_{i=1..k, 0 \notin [g_i]} (|[g_i]| r_{c_i}) + \gamma - \overline{g_a^m} - \overline{g_b^m} - \overline{g_c^m} - \alpha \times \overline{g_a^0} - \gamma - \beta \\ G(og_1) - G(og_2) & = & \overline{g_a^0} - \underline{g_b^m} - \underline{g_b^0} + \sum_{i=1..k, 0 \notin [g_i]} (|[g_i]| r_{c_i}) - \overline{g_b^m} - \overline{g_c^m} - \alpha \times \overline{g_a^0} - \beta \end{array}$$

If we use the upper bound of β:

$$\begin{array}{rcl} G(og_1) - G(og_2) & \geq & \overline{g_a^0} - \underline{g_b^m} - \underline{g_b^0} + \sum_{i=1..k, 0 \notin [g_i]} (|[g_i]| r_{c_i}) - \overline{g_b^m} - \overline{g_c^m} - \alpha \times \overline{g_a^0} - (1-\alpha)(\overline{g_a^0} - \underline{g_a^0}) - \overline{g_b^0} + \underline{g_b^0} \\ G(og_1) - G(og_2) & \geq & -\underline{g_b^m} + \sum_{i=1..k, 0 \notin [g_i]} (|[g_i]| r_{c_i}) - \overline{g_b^m} - \overline{g_c^m} + (1-\alpha)\underline{g_a^0} - \overline{g_b^0} \end{array}$$

We can write the constraints (5.4), (5.5) for the grouping og_1:

$$\underline{g_a^m} + \underline{g_a^0} \geq 0 \tag{B.4}$$

$$-\overline{g_b^m} - \overline{g_b^0} \geq 0 \tag{B.5}$$

It is trivial that if $0 \notin [g_i]$, then $|[g_i]| - (\overline{g_i} - \underline{g_i}) \geq 0$. Extending this relation we can obtain:

$$\sum_{i=1..k, 0 \notin [g_i]} (|[g_i]| r_{c_i}) - (\overline{g_c^m} - \underline{g_c^m}) \geq 0 \tag{B.6}$$

It is also trivial to verify that if $0 \notin [g_i]$, *then* $|[g_i]| - \overline{g_i} \geq 0$. *Extending this relation we can obtain:*

$$\sum_{i=1..k, 0 \notin [g_i]} (|[g_i]| r_{c_i}) - \overline{g_c^m} \geq 0 \tag{B.7}$$

According to the constraint (5.4) in the grouping og_2, *the value of* α *must satisfy:*

$$\alpha \leq \frac{-(\underline{g_a^m} + \underline{g_b^m} + \underline{g_c^m})}{\underline{g_a^0}}$$

The right side of the constraint is always positive: $\underline{g_a^m} + \underline{g_b^m} + \underline{g_c^m} = \underline{G_m} > 0$ *and* $\underline{g_a^0} < 0$. *Thus, we can identify two cases:*

- *When the right side is less than one if we set* α *to* $\frac{-(\underline{g_a^m} + \underline{g_b^m} + \underline{g_c^m})}{\underline{g_a^0}}$ *(i.e.,* $(1-\alpha)\underline{g_a^0} = \underline{g_a^0} + \underline{g_a^m} + \underline{g_b^m} + \underline{g_c^m}$*), then we obtain in (B.4):*

$$G(og_1) - G(og_2) \geq -\underline{g_b^m} + \sum_{i=1..k, 0 \notin [g_i]} (|[g_i]| r_{c_i}) - \overline{g_b^m} - \overline{g_c^m} + \underline{g_a^0} + \underline{g_a^m} + \underline{g_b^m} + \underline{g_c^m} - \overline{g_b^0}$$

$$G(og_1) - G(og_2) \geq \left(\sum_{i=1..k, 0 \notin [g_i]} (|[g_i]| r_{c_i}) - (\overline{g_c^m} - \underline{g_c^m}) \right) + (\underline{g_a^m} + \underline{g_a^0}) + (-\overline{g_b^m} - \overline{g_b^0})$$

Observe that the three expressions in parenthesis are positive according to (B.4), (B.5) and (B.6). Then $G(og_1) \geq G(og_2)$.

- *When the right side is more than one if we set* α *to* 1, *then we obtain in (B.4):*

$$G(og_1) - G(og_2) \geq -\underline{g_b^m} + \sum_{i=1..k, 0 \notin [g_i]} (|[g_i]| r_{c_i}) - \overline{g_b^m} - \overline{g_c^m} - \overline{g_b^0}$$

$$G(og_1) - G(og_2) \geq -\underline{g_b^m} + (-\overline{g_b^m} - \overline{g_b^0}) + \left(\sum_{i=1..k, 0 \notin [g_i]} (|[g_i]| r_{c_i}) - \overline{g_c^m} \right)$$

Observe that the first term in the left side $(-\underline{g_b^m})$ *must be positive to satisfy the constraint (B.5), the second and third terms are positive according to (B.5) and (B.7). Thus,* $G(og_1) \geq G(og_2)$. □

Lemma 12 *Let* P_3 *be the following linear program: find values* r_{a_i} r_{b_i} *and* r_{c_i} *for all* $i = 1..k$ *that minimize* $G_3 = \texttt{Diam}([G_0]) + \underline{g_a} - \overline{g_b} - \texttt{Diam}([g_c]) + \sum_{i=1}^{k} |[g_i]| r_{c_i}$ *subject to (5.4), (5.5),(5.6) and (5.7) (see page 102). If* $\forall i: \; r_{b_i} = 0$, *then* P_3 *is equivalent to the following fractional knapsack problem: (W.l.o.g., assume that the occurrences are sorted: only the first* k_1 *occurrences have a positive partial derivative.) Find the values* r_{a_i}, r_{c_i} *for* $i = k_1 + 1..k$ *that maximize* $G_3' = \sum_{i=k_1+1}^{k} v_i r_{a_i}$, *subject to:*

$$\begin{aligned} M - \sum_{i=k_1+1}^{k} w_i r_{a_i} &\geq 0 \\ r_{a_i} &\in [0,1] \\ r_{c_i} &= 1 - r_{a_i} \end{aligned}$$

where $v_i = -(\overline{g_i} - |[g_i]|)$, $w_i = -\underline{g_i}$ *and* $M = \sum_{i=1}^{k_1} \underline{g_i}$.

Proof 26 *The condition* $\forall i: \quad r_{b_i} = 0$ *implies that* $\overline{g_b} = 0$. *Thus, we can rewrite the objective function:*

$$G_3 = \texttt{Diam}([G_0]) + \sum_{i=1}^{k} \left(\underline{g_i} r_{a_i} + \left(-(\overline{g_i} - \underline{g_i}) + |[g_i]|\right) r_{c_i}\right)$$

as $r_{a_i} + r_{c_i} = 1$, *the function is equivalent to:*

$$G_3 = \texttt{Diam}([G_0]) + \sum_{i=1}^{k} \left(\underline{g_i} r_{a_i} + \left(-(\overline{g_i} - \underline{g_i}) + |[g_i]|\right) (1 - r_{a_i})\right)$$

$$G_3 = \texttt{Diam}([G_0]) + \sum_{i=1}^{k} \left(-(\overline{g_i} - \underline{g_i}) + |[g_i]|\right) + \sum_{i=1}^{k} \left(\underline{g_i} - \left(-(\overline{g_i} - \underline{g_i}) + |[g_i]|\right) r_{a_i}\right)$$

simplifying we finally obtain:

$$G_3 = \texttt{Diam}([G_0]) + \sum_{i=1}^{k} \left(-(\overline{g_i} - \underline{g_i}) + |[g_i]|\right) + \sum_{i=1}^{k} \left((\overline{g_i} - |[g_i]|\right) r_{a_i})$$

Observe that if $[g_i] \geq 0$, $|[g_i]| = \overline{g_i}$, *thus,* $G_3 = \texttt{Diam}([G_0]) + \sum_{i=1}^{k} \left(-(\overline{g_i} - \underline{g_i}) + |[g_i]|\right)$. *In other words, if* $[g_i] \geq 0$ *the objective function does not depend on the values of* r_{a_i} *and* r_{c_i}. *Thus we set* $r_{a_i} = 1$, *for all* i *with* $[g_i] \geq 0$ *because in this way we maximize the relaxation of the constraint (5.6):*

$$\sum_{i=1}^{k} \underline{g_i} r_{a_i} \geq 0$$

As only the first k_1 *occurrences have a positive partial derivative. We can rewrite the objective function:*

$$G_3 = \texttt{Diam}([G_0]) + \sum_{i=1}^{k} \left(-(\overline{g_i} - \underline{g_i}) + |[g_i]|\right) + \sum_{i=k_1+1}^{k} \left(\overline{g_i} - |[g_i]|\right) r_{a_i}$$

and the constraint (5.6):

$$\sum_{i=1}^{k_1} \underline{g_i} - \sum_{i=k_1+1}^{k} (-\underline{g_i}) r_{a_i} \geq 0$$

Note that $\texttt{Diam}([G_0]) + \sum_{i=1}^{k} \left(-(\overline{g_i} - \underline{g_i}) + |[g_i]|\right)$ *and* $\sum_{i=1}^{k_1} \underline{g_i}$ *are constants, thus the problem can be rewritten as the fractional knapsack problem of Lemma 12.* □

B.4 The average computation used for performing the comparison on evaluation diameters

Consider $\rho_1, ..., \rho_n$ the set of computed ratios (data set). We compute the *quasi-arithmetic mean* using the function $h(\rho) = \frac{\rho}{\rho+1}$, i.e.,

$$\overline{\rho} = h^{-1}\left(\frac{1}{n}\sum_{i=1}^{n} h(\rho_i)\right)$$

where $h^{-1}(\rho^*) = \frac{\rho^*}{1-\rho^*}$ is the inverse function of h. This method gives equal weight to each data point, i.e., it is not affected by very high or very low values of ρ_i.

Bibliography

Ait-Aoudia, S., Jegou, R., and Michelucci, D. (1993). Reduction of Constraint Systems. In *Compugraphic*. 135, 139, 140

Araya, I., Neveu, B., and Trombettoni, G. (2008a). Exploiter les sous-expressions communes dans les csp numériques. In *Journées francophones de programmation par contraintes (JFPC)*, pages 375–384. 115

Araya, I., Neveu, B., and Trombettoni, G. (2008b). Exploiting Common Subexpressions in Numerical CSPs. In *Proc. CP, LNCS 5202*, pages 342–357. 115

Araya, I., Neveu, B., and Trombettoni, G. (2009a). A New Monotonicity-Based Interval Extension Using Occurrence Grouping. In *Workshop IntCP*, pages 51–64. 99

Araya, I., Neveu, B., and Trombettoni, G. (2009b). An Interval Constraint Propagation Algorithm Exploiting Monotonicity. In *Workshop INTCP*, pages 65–83. 73

Araya, I., Trombettoni, G., and Neveu, B. (2009c). Filtering Numerical CSPs Using Well-Constrained Subsystems. In *Proc. CP, LNCS 5732*, pages 158–172. 135

Araya, I., Trombettoni, G., and Neveu, B. (2009d). Utilisation de sous-systèmes bien-contraints pour le filtrage de csp numériques. In *Journées francophones de programmation par contraintes (JFPC)*. 135

Archimedes (Before 212 BC). On the measurement of the circle. In *In Thomas L. Heath (ed.), The Works of Archimedes, Cambridge University Press, 1897 ; Dover edition, 1953*, pages 91–98. 9

Baharev, A. and Rév, E. (2009). A Complete Nonlinear System Solver Using Affine Arithmetic. In *Proc. CP, LNCS 5732*, pages 17–33. 43

Batnini, H., Michel, C., and Rueher, M. (2005). Mind The Gaps: A New Splitting Strategy for Consistency Techniques. In *Proc. CP, LNCS 3709*, pages 77–91. 60

Benhamou, F. and Goualard, F. (2004). Universally Quantified Interval Constraints. In *Proc. CP, LNCS 1894*, pages 67–82. 66

Benhamou, F., Goualard, F., Granvilliers, L., and Puget, J.-F. (1999). Revising Hull and Box Consistency. In *ICLP*, pages 230–244. 47, 49, 50, 51, 60, 77, 141, 154

Benhamou, F., McAllester, D., and Van Hentenryck, P. (1994). CLP(Intervals) revisited. In *Proc. of Logic Programming Symposium*, pages 124–138. 47, 50

Bliek, C. (1992). *Computer methods for design automation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA. 40

Bliek, C., Neveu, B., and Trombettoni, G. (1998). Using Graph Decomposition for Solving Continuous CSPs. In *Proc. CP, LNCS 1520*, pages 102–116. 135, 148

Buchberger, B. (1985). Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory. *Multidimensional Systems Theory*, pages 184–232. 1

Carrizosa, E., Hansen, P., and Messine, F. (2004). Improving Interval Analysis Bounds by Translations. *Journal of Global Optimization*, 29(2):157–172. 32

Ceberio, M. and Granvilliers, L. (2001). Solving Nonlinear Systems by Constraint Inversion and Interval Arithmetic. In *Proc. AISC, Artificial Intelligence and Symbolic Computation, LNCS 1930*, pages 127–141. 32, 132

Ceberio, M. and Granvilliers, L. (2002). Solving Nonlinear Equations by Abstraction, Gaussian Elimination, and Interval Methods. In *Proc. of FroCoS*, pages 117–131. 62

Ceberio, M. and Kreinovich, V. (2004). Greedy Algorithms for Optimizing Multivariate Horner Schemes. *SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation)*, 38(1):8–15. 32, 132

Chabert, G. (2007). *Techniques d'intervalles pour la résolution de systèmes d'équations.* PhD thesis, University of Nice-Sophia Antipolis, France. (in french). 28, 49

Chabert, G. (2009). `www.ibex-lib.org`. 26, 40, 65, 66, 85, 108, 129, 144

Chabert, G. and Jaulin, L. (2009a). Contractor Programming. *Artificial Intelligence*, 173:1079–1100. 66, 85, 108, 144

Chabert, G. and Jaulin, L. (2009b). Hull Consistency Under Monotonicity. In *Proc. CP, LNCS 5732*, pages 188–195. 51, 53, 73, 83

Chabert, G. and Jaulin, L. (2009c). QUIMPER, A Language for Quick Interval Modelling and Programming in a Bounded-Error Context. *Artificial Intelligence*, 173:1079–1100. 40, 65, 129

Chabert, G., Trombettoni, G., and Neveu, B. (2005). Box-Set Consistency for Interval-based Constraint Problems. In *SAC - 20th ACM Symposium on Applied Computing*, pages 1439–1443. 49, 60, 84

Collavizza, H., Delobel, F., and Rueher, M. (1999). Comparing Partial Consistencies. *Reliable Computing*, 5(3):213–228. 46, 116

Cruz, J. and Barahona, P. (2001). Global Hull Consistency with Local Search for Continuous Constraint Solving. In *Proc. EPIA, LNAI 2258*, pages 349–362. 138

Demidovitch, B. and Maron, I. (1973). *Éléments de calcul numérique.* Editions Mir, Moscou. 42

Domes, F. (2009). GloptLab - a Configurable Framework for Solving Continuous, Algebraic CSPs. In *IntCP, int. WS on interval analysis, constraint propagation, applications, at CP conference*, pages 1–16. 65

Dulmage, A. and Mendelsohn, N. (1958). Covering of Bipartite Graphs. *Canadian Journal of Mathematics*, 10:517–534. 139

Faltings, B. (1994). Arc-consistency for Continuous Variables. *Artif. Intelligence*, 65:363–376. 153

Flajolet, P., Sipala, P., and Steyaert, J.-M. (1990). Analytic variations on the common subexpression problem. In *Proc. Automata, Languages, and Programming, LNCS 443*, pages 220–334. 121

Freuder, E. (1982). A Sufficient Condition for Backtrack-Free Search. *J. ACM*, 29(1):24–32. 153

Gelfand, I., Kapranov, M., and Zelevinsky, A. (1994). *Discriminants, Resultants, and Multidimensional Determinants.* Birkhäuser. 1

Goldsztejn, A., Michel, C., and Rueher, M. (2009). Efficient Handling of Universally Quantified Inequalities. *Constraints*, 14(1):117–135. 73

Granvilliers, L. and Benhamou, F. (2006). Algorithm 852 : Realpaver : An Interval Solver using Constraint Satisfaction Techniques. *ACM Transactions on Mathematical Software.*, 31(1):138–156. 65

Granvilliers, L., Monfroy, E., and Benhamou, F. (2001). Symbolic-Interval Cooperation in Constraint Programming. In *Int. Symp. on Symbolic and Algebraic Computation (ISSAC), ACM*, pages 150–166. 115

Hansen, E. and Walster, G. W. (2003). *Global Optimization using Interval Analysis, Second Edition.* CRC Press. 25, 30, 40, 59

Harvey, W. and Stuckey, P. (2003). Improving Linear Constraint Propagation by Changing Constraint Representation. *Constraints*, 7:173–207. 131

Hirsch, M., Meneses, C., Pardalos, P., and Resende, M. (2007). Global Optimization by Continuous GRASP. *Opt. Lett.*, 1(2):201–212. 1

Horner, W. G. (1819). A new Method of Solving Numerical Equations of all Orders, by Continuous Approximation. *Philos. Trans. Roy. Soc. London*, 109:308–335. 31, 100

Hyvönen, E. (1992). Constraint reasoning based on interval arithmetic: The tolerance propagation approach. *Artificial Intelligence*, 58:71–112. 45

Iri, M. (1984). Simultaneous Computation of Functions, Partial Derivatives and Estimates of Rounding Errors : Complexity and Practicality. *Japan J. Appl. Math.*, (1):223–252. 18

Jaulin, J., Kieffer, M., Didrit, O., and Walter, E. (2001). *Applied Interval Analysis.* Springer. 66

Kearfott, R. B., Lakeyev, A., Rohn, J., and Kahl, P. (1996). *Rigorous Global Search : Continuous Problems.* Kluwer, Dordrecht. 3, 43, 61

Kearfott, R. B. and Novoa III, M. (1990). INTBIS, a portable interval Newton/Bisection package. *ACM Trans. on Mathematical Software*, 16(2):152–157. 59

Krawczyk, R. (1969). Newton-Algorithmen zur Besstimmung von Nullstellen mit Fehlerschranken. *Computing*, 4:187–201. 42

Kreinovich, V., Lakeyev, A., Rohn, J., and Kahl, P. (1997). *Computational complexity and feasibility of data processing and interval computations.* Kluwer. 22

Lebbah, Y., Michel, C., Rueher, M., Daney, D., and Merlet, J.-P. (2005). Efficient and Safe Global Constraints for Handling Numerical Constraint Systems. *SIAM Journal on Numerical Analysis*, 42(5):2076–2097. 43, 65, 66, 79, 140

Lhomme, O. (1993). Consistency Techniques for Numeric CSPs. In *IJCAI*, pages 232–238. 45, 54, 79, 139, 143

Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons. 163

Martinez, J. (1994). Algorithms for Solving Nonlinear Systems of Equations. In *Continuous Optimization: The State of the Art*, pages 81–108. 1

Merlet, J.-P. (2000). ALIAS: An Algorithms Library for Interval Analysis for Equation Systems. Technical report, INRIA Sophia. `www-sop.inria.fr/coprin/logiciels/ALIAS/ALIAS.html`. 3, 26, 31, 32, 42, 60, 61, 65, 73, 129

Merlet, J.-P. (2002). Optimal Design for the Micro Parallel Robot MIPS. In *Proc. ICRA, International Conference on Robotics and Automation, IEEE*, pages 1149–1154. 40, 46

Merlet, J.-P. (2007). Interval Analysis and Robotics. In *13th Int. Symp. of Robotics Research*. 66

Merlet, J.-P. (2009). `www-sop.inria.fr/coprin/ logiciels/ALIAS/Benches/benches.html`. 85, 108, 129, 146

Moore, R. (1966). *Interval Analysis*. Prentice Hall. 9, 16, 22, 23, 35

Muchnick, S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kauffmann. 115

Neumaier, A. and Shcherbina, O. (2004). Safe Bounds in Linear and Mixed-Integer Programming. In *Math. Programming A 99*, pages 283–296. 43

Neveu, B., Chabert, G., and Trombettoni, G. (2006). When Interval Analysis helps Interblock Backtracking. In *Proc. CP, LNCS 4204*, pages 390–405. 63, 135, 144

Neveu, B., Jermann, C., and Trombettoni, G. (2005). Inter-Block Backtracking: Exploiting the Structure in Continuous CSPs. In *Selected papers WS COCOS, LNCS 3478*, pages 15–30. 63, 135, 144, 147

Nielson, J. and Roth, B. (1999). On the Kinematic Analysis of Robotic Mechanisms. *Int. J. Robotics Research*, 18(12):1147–1160. 1

Ortega, J. and Rheinboldt, W. (1970). *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, New York. 42

Rall, L. (1981). *Automatic Differentiation: Techniques and Applications*. LNCS 120, Springer. 17

Ratz, D. (1994). Box-splitting Strategies for the Interval GaussSeidel Step in a Global Optimization Method. *Computing.*, 53:337–354. 60

Ratz, D. (1996). Inclusion Isotone Extended Interval Arithmetic - A Toolbox Update. Technical Report 5/1996, Institut für Angewandte Mathematik, Universität Karlsruhe. 19, 39

Régin, J.-C. (1994). A Filtering Algorithm for Constraints of Difference in CSPs. In *Proc. AAAI*, pages 362–367. 140

Rendl, A., Miguel, I., and Gent, I. P. (2009a). Common Subexpressions in Constraint Models of Planning Problems. In *Symp. SARA 2009*. 61

Rendl, A., Miguel, I., and Gent, I. P. (2009b). The Cost of Flattening with Common Subexpression Elimination. In *ModRef, WS on Constraint Modelling and Reformulation*, pages 117–131. 61

Rueher, M., Goldsztejn, A., Lebbah, Y., and Michel, C. (2008). Capabilities of Constraint Programming in Rigorous Global Optimization. In *NOLTA, int. Symp. on Nonlinear Theory and its Applications*. 43, 66

Sahinidis, N. V. and Twarmalani, M. (2002). *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming.* Kluwer, Dordrecht. 43

Sam-Haroud, D. and Faltings, B. (1996). Consistency Techniques for Continuous Constraints. *Constraints*, 1(1-2):85–118. 66

Schichl, H. and Neumaier, A. (2005). Interval Analysis on Directed Acyclic Graphs for Global Optimization. *Journal of Global Optimization*, 33(4):541–562. 3, 61, 115

Sherali, H. and Adams, W. (1999). *Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems.* Kluwer Academic Publishers. 43

Speelpenning, B. (1980). *Compiling Fast Partial Derivatives of Functions Given by Algorithms.* PhD thesis, Champaign, IL, USA. 17

Stahl, V. (1995). *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations.* PhD thesis, University of Linz, Austria. 32

Sunaga, T. (1958). Theory of Interval Algebra and its Application to Numerical Analysis. *Research Association of Applied Geometry (RAAG)*, 2:29–46. 9

Tapia, R. (1971). The Kantorovitch Theorem for Newton's Method. *American Mathematic Monthly*, 78(1):389–392. 42

Trombettoni, G. and Chabert, G. (2007). Constructive Interval Disjunction. In *Proc. CP, LNCS 4741*, pages 635–650. 56, 57, 79, 109, 141

Trombettoni, G. and Wilczkowiak, M. (2006). GPDOF - a Fast Algorithm to Decompose Underconstrained Geometric Constraint Systems: Application to 3S Modeling. *Int. J. Computational Geometry and Applications*, 16(5-6):479–512. 63

Van Hentenryck, P. and Michel, L. (2005). *Constraint-Based Local Search.* The MIT press. 65

Van Hentenryck, P., Michel, L., and Deville, Y. (1997). *Numerica : A Modeling Language for Global Optimization.* MIT Press. 3, 50, 51, 54, 61, 65, 77, 154

Van Hentenryck, P., Saraswat, V., and Deville, Y. (1994). Design, Implementation, and Evaluation of the Constraint Language CC(FD). *Journal of Logic Programming*, 37(1-3):139–164. 45

Vu, X.-H., Sam-Haroud, D., and Faltings, B. (2009a). Enhancing Numerical Constraint Propagation using Multiple Inclusion Representations. *Annals of Mathematics and Artificial Intelligence*, 55(3-4):295–354. 43

Vu, X.-H., Schichl, H., and Sam-Haroud, D. (2004). Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems. In *Proc. ICTAI, IEEE*, pages 72–81. 3, 61, 62, 115, 126, 127, 129

Vu, X.-H., Schichl, H., and Sam-Haroud, D. (2009b). Interval Propagation and Search on Directed Acyclic Graphs for Numerical Constraint Solving. *J. of Global Optimization*, 45(4):499–531. 61, 62, 115, 126, 127, 129

Warmus, M. (1956). Calculus of Approximations. *Bulletin de l'Academie Polonaise de Sciences*, 4(5):253–257. 9

Wolfram, S. (2009). `www.wolfram.com/products/mathematica/index.html`. 129

Yamamura, K. (2000). Finding all Solutions of Nonlinear Equations using Linear Combination of Functions. *Reliable Computing*, 6:105–113. 62

Young, R. C. (1931). The Algebra of Multi-valued Quantities. *Mathematische Annalen*, 104:260–290. 9